

Informix Guide to SQL

Tutorial

Informix Guide to SQL: Tutorial
Version 6.0

March 1994
Part No. 000-7598

THE INFORMIX SOFTWARE AND USER MANUAL ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE INFORMIX SOFTWARE AND USER MANUAL IS WITH YOU. SHOULD THE INFORMIX SOFTWARE AND USER MANUAL PROVE DEFECTIVE, YOU (AND NOT INFORMIX OR ANY AUTHORIZED REPRESENTATIVE OF INFORMIX) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION. IN NO EVENT WILL INFORMIX BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE SUCH INFORMIX SOFTWARE OR USER MANUAL, EVEN IF INFORMIX OR AN AUTHORIZED REPRESENTATIVE OF INFORMIX HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY. IN ADDITION, INFORMIX SHALL NOT BE LIABLE FOR ANY CLAIM ARISING OUT OF THE USE OF OR INABILITY TO USE SUCH INFORMIX SOFTWARE OR USER MANUAL BASED UPON STRICT LIABILITY OR INFORMIX'S NEGLIGENCE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU IN WHOLE OR IN PART. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS, WHICH VARY FROM STATE TO STATE.

All rights reserved. No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems—without permission of the publisher.

Published by: Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025

Published by INFORMIX® Press Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025

INFORMIX® and C-ISAM® are registered trademarks of Informix Software, Inc.

UNIX® is a registered trademark, licensed exclusively by the X/Open Company Ltd. in the United Kingdom and other countries.

X/Open® is a registered trademark of X/Open Company Ltd. in the United Kingdom and other countries.

MS® and MS-DOS® are registered trademarks of Microsoft Corporation.

("DOS" as used herein refers to MS-DOS and/or PC-DOS operating systems.)

PostScript® is a registered trademark of Adobe Systems Incorporated.

IBM® is a registered trademark and DRDA™ is a trademark of International Business Machines Corporation.

Some of the products or services mentioned in this document are provided by companies other than Informix. These products or services are identified by the trademark or servicemark of the appropriate company. If you have a question about one of those products or services, please call the company in question directly.

Documentation Team: Robert Berry, Mary Cole, Sally Cox, Signe Haugen, Geeta Karmarkar, Steven Klitzing, Judith Sherwood, Rob Weinberg, Chris Willis, Eileen Wollam

RESTRICTED RIGHTS LEGEND

The Informix software and accompanying materials are provided with Restricted Rights. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or subparagraphs (c) (1) and (2) of the Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable (and any other applicable license provisions set forth in the Government contract).

Copyright © 1981-1994 by Informix Software, Inc.

Preface

This book is a tutorial on the Structured Query Language (SQL) as it is implemented by Informix products. The *Informix Guide to SQL: Tutorial* and its companion volumes, the *Informix Guide to SQL: Reference* and the *Informix Guide to SQL: Syntax*, tell you how to create, manage, and use relational databases with Informix software tools.

Examples in this manual represent a number of Informix products and operating systems at different release levels.

You must have the following Informix software:

- An **INFORMIX-OnLine Dynamic Server** database server or an **INFORMIX-SE** database server
The database server either must be installed on your machine or on another machine to which your machine is connected over a network.
- Either an Informix application development tool, such as **INFORMIX-4GL**; or an SQL application programming interface (API), such as **INFORMIX-ESQL/C**; or the **DB-Access** database access utility, which is shipped as part of your database server.

The application development tool, SQL API, or **DB-Access** enables you to compose queries, send them to the database server, and view the results that the database server returns. You can use **DB-Access** to try out all the SQL statements described in this guide.

Summary of Chapters

The *Informix Guide to SQL: Tutorial* includes the following chapters:

- This Preface provides general information about the book and lists additional reference materials that can help you understand relational database management.
- The Introduction tells how SQL fits into the Informix family of products and books, explains how to use this book, introduces the demonstration database from which the product examples are drawn, and lists the new features for Version 6.0 of Informix database server products.

Part I – “Using Basic SQL.” This part contains introductory chapters on how to use SQL. Read these chapters first if you are new to databases and SQL.

- Chapter 1, “Introducing Database Features and SQL,” contains an overview of database terminology and defines some important terms and ideas that are used throughout the book.
- Chapter 2, “Composing Simple SELECT Statements,” begins the exploration of making simple queries to retrieve and display database data.
- Chapter 3, “Composing Advanced SELECT Statements,” discusses making advanced queries to retrieve and display database data.
- Chapter 4, “Modifying Data,” describes the statements you use to insert, delete, or update data, and introduces the concepts of database privileges, maintaining data integrity, and archiving data.
- Chapter 5, “Programming with SQL,” discusses calling the database server, retrieving rows, and embedding data.
- Chapter 6, “Modifying Data Through SQL Programs,” provides an in-depth look at INSERT, DELETE, and UPDATE statements and using these statements in SQL programs.
- Chapter 7, “Programming for a Multiuser Environment,” provides a complete discussion on concurrency, isolation levels, and locking.

Part II – “Designing and Managing Databases.” This part contains overview chapters about designing and managing databases. Read these chapters to understand the entity-relationship data model for databases and how to create and tune a database.

- Chapter 8, “Building Your Data Model,” describes the components of a data model and provides a step-by-step procedure for building one.
- Chapter 9, “Implementing Your Data Model,” tells you how to define database data types and create a database.

-
- Chapter 10, “Tuning Your Data Model,” discusses many of the details that help you set up an efficient database model, including disk storage, calculating table sizes, managing indexes, and maximizing concurrency.
 - Chapter 11, “Granting and Limiting Access to Your Database,” details how you can ensure data security by granting privileges and using stored procedures and views.
 - Chapter 12, “Understanding Informix Networking,” discusses networks and how you can best set up a database to work over a network.

Part III – “Using Advanced SQL.” This part contains advanced SQL chapters, such as how to optimize your queries, how to create and use stored procedures, and how to create and use triggers. Read these chapters if you need better performance in your database.

- Chapter 13, “Optimizing Your Queries,” defines techniques to refine and optimize your queries, introduces and explains the query Optimizer, and discusses the time costs of various operations.
- Chapter 14, “Creating and Using Stored Procedures,” describes how to create and use stored procedures.
- Chapter 15, “Creating and Using Triggers,” describes how to create and use triggers.

Informix Welcomes Your Comments

A reader-response card is provided with this manual. Please use this card to tell us what you like or dislike about this manual. To help us with future versions of the manual, please tell us about any corrections or clarifications that you would find useful. Return this card to:

Informix Software, Inc.
Technical Publications Department
4100 Bohannon Drive
Menlo Park, CA 94025

If you prefer to share your comments on-line, address your e-mail to:

doc@informix.com

Related Reading

If you want additional technical information on database management, consult the following texts by C. J. Date:

- *An Introduction to Database Systems, Volume I* (Addison-Wesley Publishing, 1990)
- *An Introduction to Database Systems, Volume II* (Addison-Wesley Publishing, 1983)

This guide assumes that you are familiar with your computer operating system. If you have limited UNIX system experience, you might want to look at your operating system manual or a good introductory text before you read this tutorial.

The following list contains some suggested texts about UNIX systems:

- *A Practical Guide to the UNIX System*, Second Edition, by M. Sobell (Benjamin/Cummings Publishing, 1989)
- *A Practical Guide to UNIX System V* by M. Sobell (Benjamin/Cummings Publishing, 1985)
- *Introducing the UNIX System* by H. McGilton and R. Morgan (McGraw-Hill Book Company, 1983)
- *UNIX for People* by P. Birns, P. Brown, and J. Muster (Prentice-Hall, 1985)

If you are interested in learning more about SQL, consider the following text:

- *Using SQL* by J. Groff and P. Weinberg (Osborne McGraw-Hill, 1990)

Table of Contents

Introduction

Informix Products That Use SQL	3
Products Included in This Manual	3
Other Useful Documentation	4
How to Use This Manual	5
Typographical Conventions	5
Example Code Conventions	5
Useful On-Line Files	6
ASCII and PostScript Error Message Files	7
The Demonstration Database	7
Creating the Demonstration Database	8
New Features in Informix Version 6.0 Products That Use SQL	9

Chapter 1

Introducing Database Features and SQL

Chapter Overview	1-3
Databases: What and Why?	1-3
The Data Model	1-3
Concurrent Use and Security	1-8
Centralized Management	1-9
Important Database Terms	1-11
The Relational Model	1-11
Structured Query Language	1-14
Standard SQL	1-14
Informix SQL and ANSI SQL	1-15
ANSI-Compliant Databases	1-15
NLS Databases	1-16
The Database Software	1-16
The Applications	1-16
The Database Server	1-16
Interactive SQL	1-17

Reports and Forms	1-17
General Programming	1-17
Applications and Database Servers	1-18
Summary	1-18

Chapter 2

Composing Simple SELECT Statements

Chapter Overview	2-3
Introducing the SELECT Statement	2-4
Some Basic Concepts	2-4
The Forms of SELECT	2-10
Special Data Types	2-11
Single-Table SELECT Statements	2-11
Selecting All Columns and Rows	2-12
Selecting Specific Columns	2-18
Using the WHERE Clause	2-28
Creating a Comparison Condition	2-29
Expressions and Derived Values	2-46
Using Functions in SELECT Statements	2-53
Using Stored Procedures in SELECT Statements	2-67
Multiple-Table SELECT Statements	2-69
Creating a Cartesian Product	2-69
Creating a Join	2-71
Some Query Shortcuts	2-79
Summary	2-84

Chapter 3

Composing Advanced SELECT Statements

Chapter Overview	3-3
Using the GROUP BY and HAVING Clauses	3-4
Using the GROUP BY Clause	3-4
Using the HAVING Clause	3-9
Creating Advanced Joins	3-11
Self-Joins	3-11
Outer Joins	3-21
Subqueries in SELECT Statements	3-32
Using ALL	3-33
Using ANY	3-34
Single-Valued Subqueries	3-36
Correlated Subqueries	3-37
Using EXISTS	3-38

	Set Operations	3-42
	Union	3-43
	Intersection	3-51
	Difference	3-53
	Summary	3-54
Chapter 4	Modifying Data	
	Chapter Overview	4-3
	Statements That Modify Data	4-3
	Deleting Rows	4-4
	Deleting a Known Number of Rows	4-4
	Inserting Rows	4-6
	Updating Rows	4-11
	Database Privileges	4-15
	Displaying Table Privileges	4-16
	Data Integrity	4-17
	Entity Integrity	4-18
	Semantic Integrity	4-18
	Referential Integrity	4-19
	Interrupted Modifications	4-22
	The Transaction	4-23
	Transaction Logging	4-24
	Specifying Transactions	4-24
	Archives and Logs	4-25
	Archiving with INFORMIX-SE	4-26
	Archiving with INFORMIX-OnLine Dynamic Server	4-26
	Concurrency and Locks	4-27
	Data Replication	4-28
	INFORMIX-OnLine Dynamic Server Data Replication	4-29
	Summary	4-29
Chapter 5	Programming with SQL	
	Chapter Overview	5-3
	SQL in Programs	5-3
	Static Embedding	5-5
	Dynamic Statements	5-5
	Program Variables and Host Variables	5-5
	Calling the Database Server	5-8
	The SQL Communications Area	5-8
	The SQLCODE Field	5-11
	The SQLERRD Array	5-12
	The SQLAWARN Array	5-12
	The SQLSTATE Value	5-12

Retrieving Single Rows	5-13
Data Type Conversion	5-14
Dealing with Null Data	5-15
Dealing with Errors	5-16
Retrieving Multiple Rows	5-19
Declaring a Cursor	5-20
Opening a Cursor	5-20
Fetching Rows	5-21
Cursor Input Modes	5-23
The Active Set of a Cursor	5-24
Using a Cursor: A Parts Explosion	5-26
Dynamic SQL	5-28
Preparing a Statement	5-29
Executing Prepared SQL	5-31
Dynamic Host Variables	5-33
Freeing Prepared Statements	5-33
Quick Execution	5-34
Embedding Data Definition Statements	5-34
Embedding Grant and Revoke Privileges	5-34
Summary	5-37

Chapter 6

Modifying Data Through SQL Programs

Chapter Overview	6-3
Using DELETE	6-3
Direct Deletions	6-4
Deleting with a Cursor	6-7
Using INSERT	6-8
Using an Insert Cursor	6-8
Rows of Constants	6-11
An Insert Example	6-12
Using UPDATE	6-14
Using an Update Cursor	6-15
Cleaning up a Table	6-16
Summary	6-17

Chapter 7

Programming for a Multiuser Environment

Chapter Overview	7-3
Concurrency and Performance	7-3
Locking and Integrity	7-3
Locking and Performance	7-3
Concurrency Issues	7-4
How Locks Work	7-5
Kinds of Locks	7-6



Lock Scope	7-6
The Duration of a Lock	7-8
Locks While Modifying	7-9
Setting the Isolation Level	7-9
Dirty Read Isolation	7-10
Committed Read Isolation	7-10
Cursor Stability Isolation	7-11
Repeatable Read Isolation	7-12
Setting the Lock Mode	7-13
Waiting for Locks	7-13
Not Waiting for Locks	7-14
Waiting a Limited Time	7-14
Handling a Deadlock	7-14
Handling External Deadlock	7-15
Simple Concurrency	7-15
Locking with Other Database Servers	7-16
Isolation While Reading	7-17
Locking Updated Rows	7-17
Hold Cursors	7-18
Summary	7-19

Chapter 8

Building Your Data Model

Chapter Overview	8-3
Why Build a Data Model	8-3
Entity-Relationship Data Model Overview	8-3
Identifying and Defining Your Principal Data Objects	8-4
Discovering Entities	8-4
Defining the Relationships	8-8
Identifying Attributes	8-14
Diagramming Your Data Objects	8-16
Translating E-R Data Objects Into Relational Constructs	8-20
Rules for Defining Tables, Rows, and Columns	8-20
Determining Keys for Tables	8-22
Resolving Your Relationships	8-26
Normalizing Your Data Model	8-29
Summary	8-33

Chapter 9

Implementing Your Data Model

Chapter Overview	9-3
Defining the Domains	9-3
Data Types	9-4
Default Values	9-19
Check Constraints	9-20



Creating the Database	9-20
Using CREATE DATABASE	9-21
Using CREATE TABLE	9-24
Using Command Scripts	9-26
Populating the Tables	9-27
Summary	9-29

Chapter 10

Tuning Your Data Model

Chapter Overview	10-3
INFORMIX-OnLine Dynamic Server Disk Storage	10-3
Chunks and Pages	10-4
Dbspaces and Blobspaces	10-4
Disk Mirroring	10-5
Databases	10-5
Tables and Spaces	10-6
Tblspaces	10-7
Extents	10-8
Reorganizing dbspaces and Tables	10-10
Calculating Table Sizes	10-12
Estimating Fixed-Length Rows	10-13
Estimating Variable-Length Rows	10-14
Estimating Index Pages	10-16
Estimating Blobpages	10-17
Locating Blob Data	10-18
Managing Indexes	10-19
Space Costs of Indexes	10-20
Time Costs of Indexes	10-20
Choosing Indexes	10-21
Duplicate Keys Slow Index Modifications	10-22
Dropping Indexes	10-24
Clustered Indexes	10-25
Denormalizing	10-26
Shorter Rows for Faster Queries	10-26
Expelling Long Strings	10-26
Splitting Wide Tables	10-28
Splitting Tall Tables	10-29
Redundant and Derived Data	10-30
Maximizing Concurrency	10-32
Easing Contention	10-32
Rescheduling Modifications	10-33
Isolating and Dispersing Updates	10-35
Summary	10-36

Chapter 11

Granting and Limiting Access to Your Database

- Chapter Overview 11-3
- Controlling Access to Databases 11-4
 - Securing Database Files 11-4
 - Securing Confidential Data 11-5
- Granting Privileges 11-5
 - Database-Level Privileges 11-5
 - Ownership Rights 11-7
 - Table-Level Privileges 11-7
 - Procedure-Level Privileges 11-12
 - Automating Privileges 11-13
- Controlling Access to Data Using Stored Procedures 11-15
 - Restricting Reads of Data 11-16
 - Restricting Changes to Data 11-16
 - Monitoring Changes to Data 11-17
 - Restricting Object Creation 11-18
- Using Views 11-19
 - Creating Views 11-20
 - Modifying Through a View 11-23
- Privileges and Views 11-27
 - Privileges When Creating a View 11-27
 - Privileges When Using a View 11-28
- Summary 11-30

Chapter 12

Understanding Informix Networking

- Chapter Overview 12-3
- What Is a Network? 12-4
- Database Management System Configurations 12-4
 - A Single-User Configuration 12-4
 - A Local Multiuser Configuration 12-6
 - A Remote Configuration 12-8
 - Single-Machine Configuration Using Network Communication 12-9
 - Distributed Databases 12-10
 - Distributed Databases Using Multiple Vendor Servers 12-11
- Connecting to Data on a UNIX Network 12-12
 - Example of Client / Server Connections 12-13
 - Environment Variables 12-14
 - Connection Information 12-15
 - SQL Connection Statements 12-16
- Accessing Tables 12-16
 - Using Synonyms with Table Names 12-17
 - Synonym Chains 12-18
- Protecting Your Data in a Networked Environment 12-19

Data Protection with INFORMIX-SE	12-19
Data Protection with INFORMIX-OnLine Dynamic Server	12-19
Data Integrity for Distributed Data	12-20
Summary	12-21

Chapter 13

Optimizing Your Queries

Chapter Overview	13-3
Optimizing Techniques	13-4
Verifying the Problem	13-4
Considering the Total System	13-5
Understanding the Application	13-5
Measuring the Application	13-6
Finding the Guilty Functions	13-7
Keeping an Open Mind	13-7
The Query Optimizer	13-8
How the Optimizer Works	13-8
Reading the Plan	13-12
Time Costs of a Query	13-14
Activities in Memory	13-14
Disk-Access Management	13-14
The Cost of Reading a Row	13-16
The Cost of Sequential Access	13-17
The Cost of Nonsequential Access	13-18
The Cost of Rowid Access	13-18
The Cost of Indexed Access	13-18
The Cost of Small Tables	13-19
The Cost of NLS Functionality	13-19
The Cost of Network Access	13-20
The Importance of Table Order	13-22
Making Queries Faster	13-27
Preparing a Test Environment	13-28
Studying the Data Model	13-28
Studying the Query Plan	13-29
Examining the Indexes	13-29
Refining the Column Filters	13-31
Rethinking the Query	13-33
Reviewing the Optimization Level	13-36
Using a Temporary Table to Speed Queries	13-36
Summary	13-41

Chapter 14

Creating and Using Stored Procedures

- Chapter Overview 14-3
- Introduction to Stored Procedures and SPL 14-3
 - What You Can Do with Stored Procedures 14-3
 - Relationship Between SQL and a Stored Procedure 14-4
- Creating and Using Stored Procedures 14-4
 - Creating a Procedure Using DB-Access 14-5
 - Creating a Procedure Using an SQL API 14-5
 - Commenting and Documenting a Procedure 14-6
 - Diagnosing Compile-Time Errors 14-6
 - Looking at Compile-Time Warnings 14-8
 - Generating the Text or Documentation 14-8
 - Executing a Procedure 14-9
 - Executing a Stored Procedure Dynamically 14-11
 - Debugging a Procedure 14-11
 - Re-creating a Procedure 14-13
- Privileges on Stored Procedures 14-13
 - Privileges Necessary at Creation 14-14
 - Privileges Necessary at Execution 14-14
 - Revoking Privileges 14-16
- Variables and Expressions 14-16
 - Variables 14-16
 - SPL Expressions 14-21
- Program Flow Control 14-22
 - Branching 14-22
 - Looping 14-23
 - Function Handling 14-24
- Passing Information into and out of a Procedure 14-25
 - Returning Results 14-25
- Exception Handling 14-27
 - Trapping an Error and Recovering 14-28
 - Scope of Control of an ON EXCEPTION Statement 14-29
 - User-Generated Exceptions 14-30
- Summary 14-33

Chapter 15

Creating and Using Triggers

- Chapter Overview 15-3
- When to Use Triggers 15-3
- How to Create a Trigger 15-4
 - Assigning a Trigger Name 15-5
 - Specifying the Trigger Event 15-5
 - Defining the Triggered Actions 15-6
 - A Complete CREATE TRIGGER Statement 15-6

Using Triggered Actions	15-7
Using BEFORE and AFTER Triggered Actions	15-7
Using FOR EACH ROW Triggered Actions	15-8
Using Stored Procedures as Triggered Actions	15-10
Tracing Triggered Actions	15-12
Generating Error Messages	15-14
Applying a Fixed Error Message	15-15
Generating a Variable Error Message	15-16
Summary	15-17

Index

Introduction

Informix Products That Use SQL	3
Products Included in This Manual	3
Other Useful Documentation	4
How to Use This Manual	5
Typographical Conventions	5
Example Code Conventions	5
Useful On-Line Files	6
ASCII and PostScript Error Message Files	7
The Demonstration Database	7
Creating the Demonstration Database	8
Compliance with Industry Standards	9
New Features in Informix Version 6.0 Products That Use SQL	10



Structured Query Language (SQL) is an English-like language that you can use when creating, managing, and using relational databases. The SQL provided with Informix products is an enhanced version of the industry-standard query language developed by International Business Machines Corporation (IBM).

Informix Products That Use SQL

Informix produces many application development tools and SQL application programming interfaces (API). Application development tools currently available include products like **INFORMIX-SQL**, **INFORMIX-4GL**, and the **Interactive Debugger**. SQL APIs currently available include **INFORMIX-ESQL/C** and **INFORMIX-ESQL/COBOL**.

Informix products work with a database server, either **INFORMIX-OnLine Dynamic Server** or **INFORMIX-SE**. The **DB-Access** database access utility is shipped as a part of each database server.

If you are running client applications developed with Version 4.1 and 5.0 application development tools, you use **INFORMIX-NET** to connect the client to the network.

Products Included in This Manual

All of the information presented in this manual is valid for the following products:

- **INFORMIX-ESQL/C**, Version 6.0
- **INFORMIX-ESQL/COBOL**, Version 6.0
- **INFORMIX-SE**, Version 6.0
- **INFORMIX-OnLine Dynamic Server**, Version 6.0
- **INFORMIX-OnLine/Optical**, Version 6.0
- **INFORMIX-TP/XA**, Version 6.0

An additional product that uses information from this manual is the **INFORMIX-Gateway with DRDA**, Version 6.0.

In addition, some examples in this manual are built using Informix products at earlier release levels.

Other Useful Documentation

You can refer to a number of related Informix documents that complement this manual.

- A companion volume to the Tutorial, the *Informix Guide to SQL: Reference*, provides reference information on the types of Informix databases you can create, the data types supported in Informix products, system catalog tables associated with the database, environment variables, and the SQL utilities. This guide also provides a detailed description of the **stores6** demonstration database and contains a glossary.
- An additional companion volume to the Tutorial, the *Informix Guide to SQL: Syntax*, provides a detailed description for all of the SQL statements supported by Informix products. This guide also provides a detailed description of stored procedure language (SPL) statements.
- The *SQL Quick Syntax Guide* contains syntax diagrams for all statements and segments described in this manual.
- You, or whoever installs your Informix products, should refer to the *UNIX Products Installation Guide* for your particular release to ensure that your Informix product is properly set up before you begin to work with it. A matrix depicting possible client/server configurations is included in the *Installation Guide*.
- Depending on the database server you are using, you or your system administrator need either the *INFORMIX-SE Administrator's Guide* or the *INFORMIX-OnLine Dynamic Server Administrator's Guide* and *INFORMIX-OnLine Dynamic Server Archive and Backup Guide*.
- The *DB-Access User Manual* describes how to invoke the utility to access, modify, and retrieve information from Informix database servers.
- When errors occur, you can look them up by number and learn their cause and solution in the *Informix Error Messages* manual. If you prefer, you can look up the error messages in the on-line message file described in the section "ASCII and PostScript Error Message Files" later in this Introduction.


How to Use This Manual

This manual assumes that you are using **INFORMIX-OnLine Dynamic Server** as your database server. Features and behavior specific to **INFORMIX-SE** are noted throughout the manual.

Other conventions used in this manual for typographical format and example code are described in the following sections.

Typographical Conventions

Informix product manuals use a standard set of conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth. The following typographical conventions are used throughout this manual:

<i>italics</i>	New terms, emphasized words, and variables are printed in italics.
boldface	Database names, table names, column names, filenames, utilities, and other similar terms are printed in boldface.
<code>computer</code>	Information that the product displays and information that you enter are printed in a computer typeface.
KEYWORD	All keywords appear in uppercase letters.
	This symbol indicates a warning. Warnings provide critical information that, if ignored, could cause harm to your database.

Additionally, when you are instructed to “enter” or “execute” text, immediately press RETURN after the entry. When you are instructed to “type” the text or “press” a key, no RETURN is required.

Example Code Conventions

Examples of SQL code occur throughout this manual. Except where noted, the code is not specific to any single Informix application development tool. If only SQL statements are listed in the example, they are not delineated by semicolons. To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using the Query-language option of **DB-Access**, you must delineate multiple statements with semicolons. If you are using an SQL API, you must use EXEC SQL and a semi-colon (or other appropriate delimiters) at the start and end of each statement, respectively.

For example, you might see the following example code:

```
CONNECT TO stores6
.
.
.
DELETE FROM customer
      WHERE customer_num = 121
.
.
.
COMMIT WORK
DISCONNECT CURRENT
```

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the manual for your product.

Also note that dots in the example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.

Useful On-Line Files

In addition to the Informix set of manuals, the following on-line files, located in the `$INFORMIXDIR/release` directory, may supplement the information in this manual:

- | | |
|----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Documentation Notes | describe features not covered in the manual or that have been modified since publication. The file containing the Documentation Notes for this product is called SQLTDOC_6.0 . |
| Release Notes | describe feature differences from earlier versions of Informix products and how these differences may affect current products. The file containing the Release Notes for Version 6.0 of Informix database server products is called SERVERS_6.0 . |
| Machine Notes | describe any special actions required to configure and use Informix products on your machine. Machine notes are named for the product described, for example, the machine notes file for INFORMIX-OnLine Dynamic Server is ONLINE_6.0 . |

Please read these files because they contain vital information about application and performance issues.

A number of Informix products also provide on-line Help files that walk you through each menu option. To invoke the Help feature, simply press CTRL-W wherever you are in your Informix product.

ASCII and PostScript Error Message Files

Informix software products provide ASCII files that contain all the Informix error messages and their corrective actions. To access the error messages in the ASCII file, Informix provides scripts that let you display error messages on the screen (**finderr**) or print formatted error messages (**rofferr**). See the Introduction to the *Informix Error Messages* manual for a detailed description of these scripts.

The optional **Informix Messages and Corrections** product provides PostScript files that contain the error messages and their corrective actions. If you have installed this product, you can print the PostScript files on a PostScript printer. The PostScript error messages are distributed in a number of files of the format **errmsg1.ps**, **errmsg2.ps**, and so on. These files are located in the **\$INFORMIXDIR/msg** directory.

The Demonstration Database

The **DB-Access** utility, which is provided with your Informix database server products, includes a demonstration database called **stores6** that contains information about a fictitious wholesale sporting-goods distributor. The sample command files that make up a demonstration application are also included.

Most of the examples in this manual are based on the **stores6** demonstration database. The **stores6** database is described in detail and its contents are listed in Appendix A of the *Informix Guide to SQL: Reference*.

The script that you use to install the demonstration database is called **dbaccessdemo6** and is located in the **\$INFORMIXDIR/bin** directory. The database name that you supply is the name given to the demonstration database. If you do not supply a database name, the name defaults to **stores6**. Follow these rules for naming your database:

- Names for databases can be up to 18 characters long for **INFORMIX-OnLine Dynamic Server** databases and up to 10 characters long for **INFORMIX-SE** databases.
- The first character of a name must be a letter or an underscore (_).

- You can use letters, characters, and underscores (_) for the rest of the name.
- **DB-Access** makes no distinction between uppercase and lowercase letters.
- The database name should be unique.

When you run **dbaccessdemo6**, you are, as the creator of the database, the owner and Database Administrator (DBA) of that database.

If you installed your Informix database server according to the installation instructions, the files that make up the demonstration database are protected so you cannot make any changes to the original database.

You can run the **dbaccessdemo6** script again whenever you want to work with a fresh demonstration database. The script prompts you when the creation of the database is complete, and asks if you would like to copy the sample command files to the current directory. Enter “N” if you have made changes to the sample files and do not want them replaced with the original versions. Enter “Y” if you want to copy over the sample command files.

Creating the Demonstration Database

Use the following steps to create and populate the demonstration database:

1. Set the INFORMIXDIR environment variable so that it contains the name of the directory in which your Informix products are installed. Set INFORMIXSERVER to the name of the default database server. The name of the default database server must exist in the **\$INFORMIXDIR/etc/sqlhosts** file. (For a full description of environment variables, see Chapter 4 of the *Informix Guide to SQL: Reference*.) For more information about **sqlhosts**, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide* or *INFORMIX-SE Administrator's Guide*.
2. Create a new directory for the SQL command files. Create the directory by entering

```
mkdir dirname
```
3. Make the new directory the current directory by entering

```
cd dirname
```


4. Create the demonstration database and copy over the sample command files by entering the following command:

To create the database without logging enter:

```
dbaccessdemo6 dbname
```

To create the demonstration database with logging enter:

```
dbaccessdemo6 -log dbname
```

If you are using **INFORMIX-OnLine Dynamic Server**, by default the data for the database is put into the root dbspace. If you wish, you can specify a dbspace for the demonstration database.

To create a demonstration database in a particular dbspace enter:

```
dbaccessdemo6 dbspacename
```

If you are using **INFORMIX-SE**, a subdirectory called **dbname.dbs** is created in your current directory and the database files associated with **stores6** are placed there. You will see both data (**.dat**) and index (**.idx**) files in the **dbname.dbs** directory. (If you specify a dbspace name, it will be ignored.)

To use the database and the command files that have been copied to your directory, you must have UNIX read and execute permissions for each directory in the pathname of the directory from which you ran the **dbaccessdemo6** script.

5. To give someone else the permissions to access the command files in your directory, use the UNIX **chmod** command.
6. To give someone else the access to the database that you have created, grant them the appropriate privileges using the **GRANT** statement. To remove privileges, use the **REVOKE** statement. The **GRANT** and **REVOKE** statements are described in Chapter 1 of the *Informix Guide to SQL: Syntax*.

New Features in Informix Version 6.0 Products That Use SQL

The Introduction to each Version 6.0 product manual contains a list of new features for that product. The Introduction to each manual in the Version 6.0 *Informix Guide to SQL* series contains a list of new SQL features.

A comprehensive listing of all the new features for Version 6.0 Informix products is found in the Release Notes file called **SERVERS_6.0**.

This section highlights the major new features implemented in Version 6.0 of Informix products that use SQL.

- Native Language Support

Native Language Support (NLS) makes Informix Version 6.0 products adaptable to various European cultural and language environments without requiring changes to the application source code.

When appropriate environment variables are set to activate NLS and specify a locale, Informix products can properly collate strings that contain foreign characters, accept money and decimal input, and print dates and times in the format required by the locale where the software is run.

The user can

- Create or access database information in any language available on the system by changing a few environment variables
- Name user-defined objects such as databases, tables, columns, views, cursors, and files using a foreign character set
- Use the new NCHAR and NVARCHAR data types in place of CHAR and VARCHAR, respectively, for storing national characters

In addition, by installing one or more language supplements with an Informix product, the user can view error and warning messages in the language of the locale.

- Enhanced Database Connections

You can now use three new statements, CONNECT TO, DISCONNECT, and SET CONNECTION to provide a *connection-oriented* association between client and server processes in a networked or a non-networked environment. These three statements are compliant with X/Open and ANSI/ISO specifications. Applications with embedded SQL can use these statements for more uniform and portable syntax when accessing local or remote data.

- Cascading Deletes

Support for cascading deletes is provided in **INFORMIX-OnLine Dynamic Server** as an enhancement to referential integrity. Previously, when you attempted to delete a row from a parent table without deleting rows from associated child tables first, the delete was disallowed. Now you can specify the ON DELETE CASCADE option on either the CREATE TABLE or ALTER TABLE statements or on **DB-Access** menus to allow deletes from a parent table to cause deletes on child tables.

- Enhanced CREATE INDEX statement

You can now use the FILLFACTOR option on the CREATE INDEX statement to create indexes that provide for expansion of the index at a later time or to create compacted indexes.

- Enhanced Arithmetic Functions

You can now use certain trigonometric and algebraic functions in your SQL statements. The new functions include ABS, MOD, POW, ROOT, SQRT, COS, SIN, TAN, ACOS, ASIN, ATAN, ATAN2, EXP, LOGN, and LOG10.

- Enhanced Error Handling

You can now use a new statement to retrieve standard and multiple diagnostic information about your SQL statements. The GET DIAGNOSTICS statement is compliant with X/Open and ANSI/ISO specifications and provides a standard method for detecting and handling error messages. The GET DIAGNOSTICS statement is used with a new error-handling status variable called SQLSTATE.

- New DBINFO Function

The DBINFO function allows you to find the following information:

- The dbspace name for a given table
- The last serial value inserted into a table
- The number of rows processed by SELECT, INSERT, DELETE, UPDATE, and EXECUTE PROCEDURE statements

- New Environment Variables

The following Informix environment variables and X/Open categories, described in Chapter 4 of the *Informix Guide to SQL: Reference*, are new in Version 6.0:

- ARC_DEFAULT
- ARC_KEYPAD
- COLLCHAR
- DBAPICODE
- DBNLS
- DBSPACETEMP
- DBUPSPACE
- ENVIGNORE
- INFORMIXC

- INFORMIXSERVER
- INFORMIXSHMBASE
- INFORMIXSTACKSIZE
- LANG
- LC_COLLATE
- LC_CTYPE
- LC_MONETARY
- LC_NUMERIC
- LC_TIME

- New Data Distribution Features

You can use the UPDATE STATISTICS statement to create data distributions for each table. The database server uses these data distributions to improve the choice of execution paths for SELECT statements. The new data-distribution feature affects application development in the following three ways:

- The syntax for the UPDATE STATISTICS statement has been expanded.
- A new environment variable, DBUPSPACE, sets the upper limit of disk space that you want to use when columns are sorted.
- You can use the **dbschema** utility to print distribution information.

- Introduction of Environment-Configuration Files

The following environment-configuration files can contain default values of environment variables:

- \$INFORMIXDIR/etc/informix.rc
- ~/.informix

These optional files allow the administrator or the user to set environment variables in much the same way a **.login** or **.profile** file is used, so that the variables do not need to be set at each session.



Section I: Using Basic SQL





Introducing Database Features and SQL

Chapter Overview 3

Databases: What and Why? 3

The Data Model 3

Storing Data 6

Querying Data 6

Modifying Data 8

Concurrent Use and Security 8

Centralized Management 9

Group and Private Databases 10

Essential Databases 10

Important Database Terms 11

The Relational Model 11

Tables 11

Columns 12

Rows 12

Tables, Rows, and Columns 12

Operations on Tables 13

Structured Query Language 14

Standard SQL 14

Informix SQL and ANSI SQL 15


ANSI-Compliant Databases 15

NLS Databases 16

The Database Software 16

The Applications 16

The Database Server 16



Interactive SQL	17
Reports and Forms	17
General Programming	17
Applications and Database Servers	18
Summary	18

Chapter Overview

This is a book about databases and about how you can exploit them using Informix software. Your real use of a database begins with the SELECT statement, which is described in Chapter 2, “Composing Simple SELECT Statements.” If you are in a hurry, and if you know at least a little about databases, turn to it now.

This chapter covers the fundamental concepts of databases and defines some terms that are used throughout the book, emphasizing the following topics:

- What makes a database different from a collection of files?
- What terms are used to describe the main components of a database?
- What language is used to create, query, and modify a database?
- What are the main parts of the software that manages a database, and how do these parts work with each other?

Databases: What and Why?

A database is a collection of information—but so is a simple computer file. What makes a database so special? There must be reasons; otherwise people would not spend so much money, effort, and computer time on databases. Two fundamental differences exist. First, a database comprises not only data but also a plan, or *model*, of the data. Second, a database can be a common resource, used concurrently by many people.

The Data Model

The principal difference between information collected in a database and information in a file is in the way the data is organized. A flat file is organized physically; certain items precede or follow other items. But the contents of a database are organized according to a *data model*. A data model is a plan, or map, that defines the units of data and specifies how each unit is related to the others.

For example, a number can appear in either a file or a database. In a file, it is simply a number that occurs at a certain point in the file. A number in a database, however, has a role assigned to it by the data model. It may be a *price* that is associated with a *product* that was sold as one *item* of an *order* that was placed by a *customer*. Each of these things—prices, products, items, orders, and customers—are also roles specified by the data model. (See Figure 1-1.)

The data model is designed when the database is created; then units of data are inserted according to the plan laid out in the model. In some books, the term *schema* is used instead of *data model*.

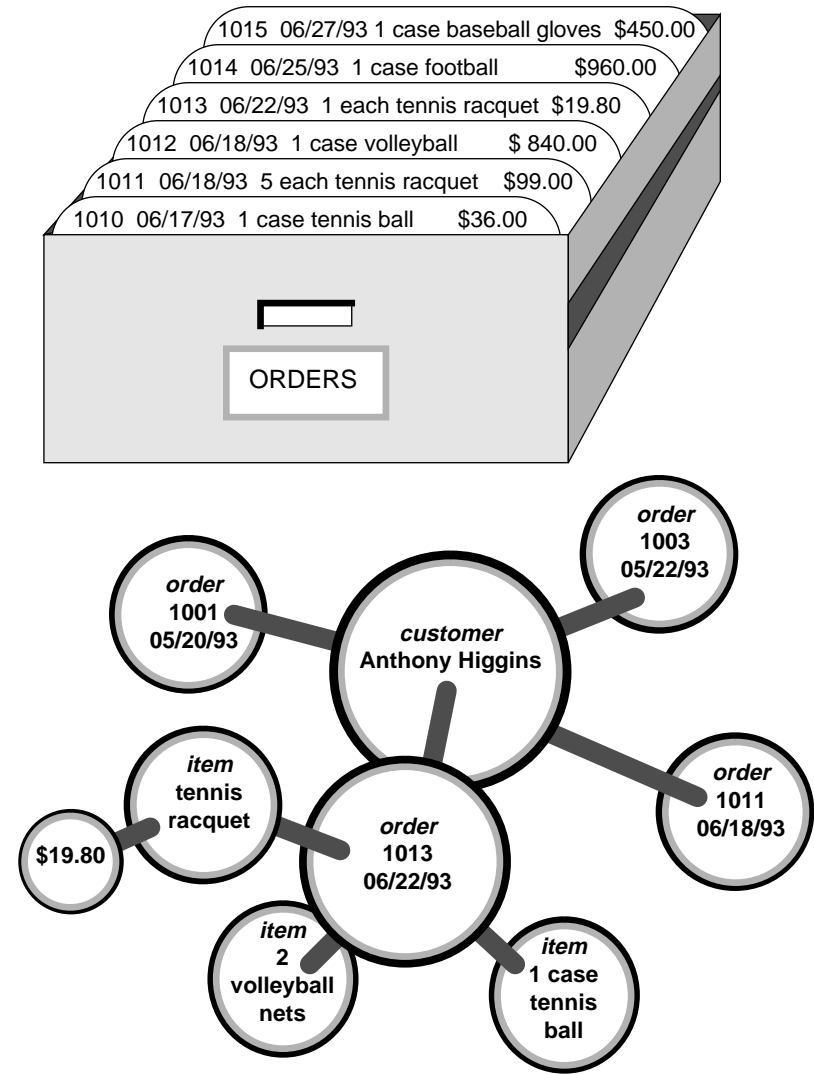


Figure 1-1 Illustration of a data model

Storing Data

Another difference between a database and a file is that the organization of the database is stored with the database.

A file may have a complex inner structure, but the definition of that structure is not within the file; it is in the programs that create or use the file. For example, a document file as stored by a word-processing program may contain very detailed structures describing the format of the document. However, only the word-processing program can decipher the contents of the file because the structure is defined within the program, not within the file.

A data model, however, is contained in the database it describes. It travels with the database and is available to any program that uses the database. The model defines not only the names of the data items but also their data types, so a program can adapt itself to the database. For example, a program can find out that, in the current database, a *price* item is a decimal number with eight digits, two to the right of the decimal point; then it can allocate storage for a number of that type. How programs work with databases is the subject of Chapter 5, “Programming with SQL,” and Chapter 6, “Modifying Data Through SQL Programs.”

Querying Data

Another difference between a database and a file is the way you can interrogate them. You can search a file sequentially, looking for particular values at particular physical locations in each line or record. That is, you might ask a file, “What records have numbers under 20 in the fifth field?” Figure 1-2 illustrates this type of search.

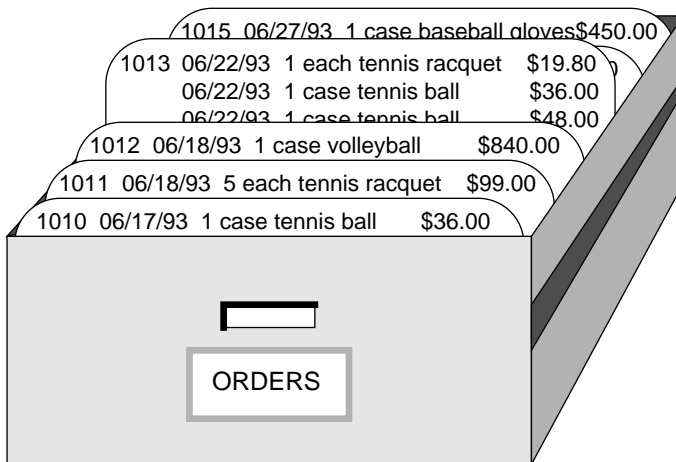


Figure 1-2 Searching a file sequentially

In contrast, when you query a database you use the terms defined by its model. To a database, you can pose questions such as, “What *orders* have been placed for *products* made by the Shimara Corporation, by *customers* in New Jersey, with *ship dates* in the third quarter?” Figure 1-3 illustrates this type of query.

In other words, when you interrogate data stored in a file, you must state your question in terms of the physical layout of the file. When you query a database, you can ignore the arcane details of computer storage and state your query in terms that reflect the real world—at least, to the extent that the data model reflects the real world.

In this manual, Chapters 2 and 3 discuss the language you use for making queries. Chapters 8 through 11 discuss designing an accurate, robust data model for other users to query.

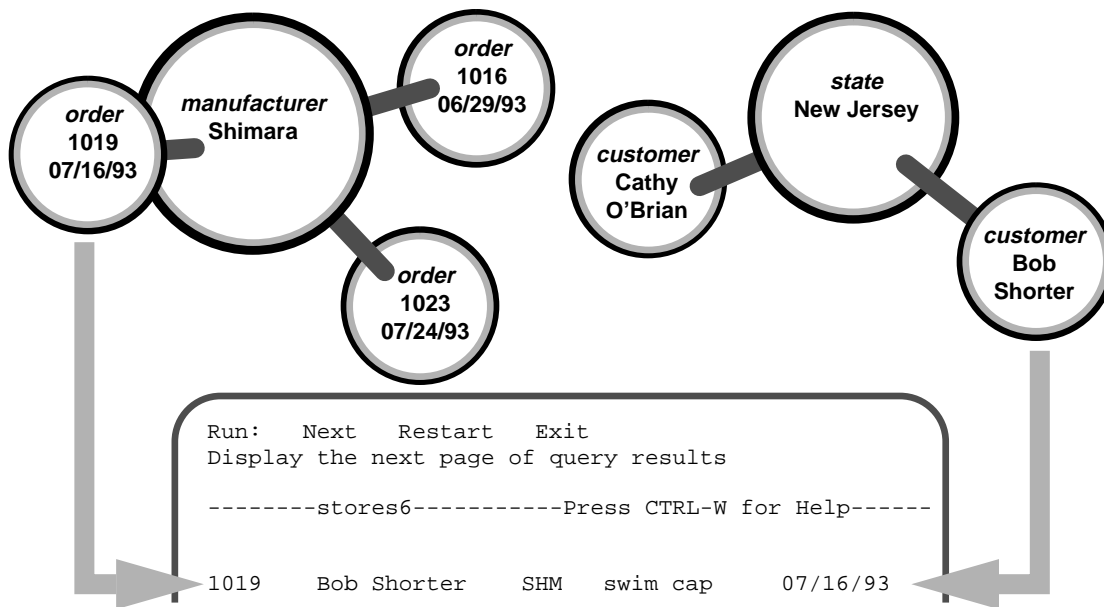


Figure 1-3 Querying a database

Modifying Data

The model also makes it possible to modify the contents of the database with less chance for error. You can query the database with commands such as “Find every *stock item* with a *manufacturer* of Presta or Schraeder, and increase its *price* by 13 percent.” You state changes in terms that reflect the meaning of the data. You do not have to waste time and effort thinking about details of fields within records in a file, so the chances for error are less.

The statements you use to modify stored data are covered in Chapter 5, “Programming with SQL.”

Concurrent Use and Security

A database can be a common resource for many users. Multiple users can query and modify a database simultaneously. The database *server* (the program that manages the contents of all databases) ensures that the queries and modifications are done in sequence and without conflict.

Having concurrent users on a database provides great advantages but also introduces new problems of security and privacy. Some databases are private; individuals set them up for their own use. Other databases contain confidential material that must be shared but among only a select group of persons. Still other databases provide public access.

Informix database software provides the means to control database use. When you design a database, you can perform any of the following functions:

- Keep the database completely private.
- Open its entire contents to all users or to selected users.
- Restrict the selection of data that some users can view. (In fact, you can reveal entirely different selections of data to different groups of users.)
- Allow specified users to view certain items but not modify them.
- Allow specified users to add new data but not modify old data.
- Allow specified users to modify all, or specified items of, existing data.
- Ensure that added or modified data conforms to the data model.

The facilities that make these and other things possible are discussed in Chapter 11, “Granting and Limiting Access to Your Database.”

Centralized Management

Databases that are used by many people are highly valuable and must be protected as important business assets. This creates two significant problems: protecting data and maintaining performance. The **INFORMIX-OnLine Dynamic Server** database server allows you to centralize these tasks.

Databases must be guarded against loss or damage. The hazards are many: failures in software and hardware, and the risks of fire, flood, and other natural disasters. Losing an important database creates a huge potential for damage. The damage could include not only the expense and difficulty of recreating the lost data but also the loss of productive time by the database users as well as the loss of business and good will while users cannot work. A plan for regular archiving of critical databases can help avoid or mitigate these potential disasters.

A large database used by many people must be maintained and tuned. Someone must monitor its use of system resources, chart its growth, anticipate bottlenecks, and plan for expansion. Users will report problems in the application programs; someone must diagnose these and correct them. If rapid response is important, someone must study and analyze the performance of the system and find the causes of slow responses.

Group and Private Databases

Some Informix database servers are designed to manage relatively small databases that are used privately by individuals or that are shared among a small group of users.

These database servers (for example, **INFORMIX-SE** for the UNIX operating system) store databases in files managed by the host operating system. These databases can be archived using the same procedures for backing up files that work with other computer files; that is, copying the files to another medium when they are not in use. The only difference is that when a database is archived its transaction log file must be reset to empty. (The use of transaction logs is discussed in Chapter 7, “Programming for a Multiuser Environment.” Chapter 10, “Tuning Your Data Model,” has more information on archiving.)

Performance problems that arise in group and private databases are usually related to particular queries that take too long. Chapter 13, “Optimizing Your Queries,” deals in depth with the reasons why a query takes more or less time. After you understand all the features of the **SELECT** statement and the alternative ways of stating a query, as covered in Chapter 2, “Composing Simple **SELECT** Statements,” and Chapter 3, “Composing Advanced **SELECT** Statements,” you can use the tips in Chapter 13, “Optimizing Your Queries,” to improve the performance of queries that otherwise might take excessive amounts of time.

Essential Databases

The **INFORMIX-OnLine Dynamic Server** database server is designed to manage large databases with requirements for high reliability, high availability, and high performance. Although it supports private and group databases very well, it is at its best managing the databases that are essential for your organization to carry out its work.

INFORMIX-OnLine Dynamic Server gives you the ability to make archival copies while the databases are in use. It also allows incremental archiving (archiving only modified data), an important feature when making a complete copy could take many reels of tape.

INFORMIX-OnLine Dynamic Server has an interactive monitor program by which its operator (or any user) can monitor the activities within the database server to see when bottlenecks are developing. It also comes with utility programs to analyze its use of disk storage. In addition, **OnLine** provides the **sysmaster** tables that contain information about an entire **OnLine** database server, which might manage many databases. For more information about the **sysmaster** tables, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

Chapter 10, “Tuning Your Data Model,” contains an overview of the **INFORMIX-OnLine Dynamic Server** disk-storage methods as part of the necessary background information to the main topic. All the details of using and managing **OnLine** are contained in the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

Important Database Terms

You should know two sets of terms before beginning the next chapter. One set of terms describes the database and the data model; the other set describes the computer programs that manage the database.

The Relational Model

Informix databases are *relational* databases. In technical terms, that means that the data model by which an Informix database is organized is based on the relational calculus devised by E.F. Codd. In practical terms, it means that all data is presented in the form of *tables* comprising *rows* and *columns*.

Tables

A database is a collection of information grouped into one or more tables. A table is an array of data *items* organized into rows and columns. A demonstration database is distributed with every Informix product. A partial table from the demonstration database is shown in Figure 1-4.

stock Table					
stock_num	manu_code	description	unit_price	unit	unit_descr
1	HRO	baseball gloves	250.00	case	10 gloves/case
1	HSK	baseball gloves	800.00	case	10 gloves/case
1	SMT	baseball gloves	450.00	case	10 gloves/case
2	HRO	baseball	126.00	case	24/case
3	HSK	baseball bat	240.00	case	12/case
4	HSK	football	960.00	case	24/case
4	HRO	football	480.00	case	24/case
5	NRG	tennis racquet	28.00	each	each
...
313	ANZ	swim cap	60.00	box	12/box

Figure 1-4 *The stock table from the demonstration database distributed with all Informix products*

A table represents all that is known about one entity, one type of thing that the database describes. The example table, **stock**, represents all that is known about the merchandise that is stocked by a sporting-goods store. Other tables in the demonstration database represent such entities as **customer** and **orders**.

A database can be thought of as a collection of tables. To create a database is to create a set of tables. The right to query or modify tables can be controlled on a table-by-table basis, so that some users can view or modify some tables and not others.

Columns

Each column of a table stands for one *attribute*, which is one characteristic, feature, or fact that is true of the subject of the table. The **stock** table has columns for the following facts about items of merchandise: stock numbers, manufacturer codes, descriptions, prices, and units of measure.

Rows

Each row of a table stands for one *instance* of the subject of the table, which is, one particular example of that entity. Each row of the **stock** table stands for one item of merchandise that is sold by the sporting-goods store.

Tables, Rows, and Columns

You now understand that the relational model of data is a very simple way of organizing data to reflect the world using the following simple corresponding relationships:

table = entity	A table represents all that the database knows about one subject or kind of thing.
column = attribute	A column represents one feature, characteristic, or fact that is true of the table subject.
row = instance	A row represents one individual instance of the table subject.

Some rules apply about how you choose entities and attributes, but they are important only when you are designing a new database. (Database design is covered in Chapters 8 through 11.) The data model in an existing database already is set. To use the database, you need to know only the names of the tables and columns and how they correspond to the real world.

Operations on Tables

Because a database is really a collection of tables, database operations are operations on tables. The relational model supports three fundamental operations, two of which are illustrated in Figure 1-5. (All three operations are defined in more detail, with many examples, in Chapter 2, “Composing Simple SELECT Statements,” and Chapter 3, “Composing Advanced SELECT Statements.”)

To *select* from a table is to choose certain rows, leaving others aside. One selection that could be made on the **stock** table is “select all rows in which the manufacturer code is HRO and the unit price is between 100.00 and 200.00.”

To *project* from a table is to choose certain columns, leaving others aside. One projection that can be made from the **stock** table is “show me the **stock_num**, **unit_descr**, and **unit_price** columns only.”

A table contains information about only one entity; when you want information about multiple entities, you must *join* their tables. You can join tables in many ways. (The join operation is the subject of Chapter 3, “Composing Advanced SELECT Statements.”)

stock Table

stock_num	manu_code	description	unit_price	unit	unit_descr
1	HRO	baseball gloves	250.00	case	10 gloves/case
1	HSK	baseball gloves	800.00	case	10 gloves/case
1	SMT	baseball gloves	450.00	case	10 gloves/case
2	HRO	baseball	126.00	case	24/case
3	HSK	baseball bat	240.00	case	12/case
4	HSK	football	960.00	case	24/case
4	HRO	football	480.00	case	24/case

P R O J E C T I O N

SELECTION

Figure 1-5 Illustration of selection and projection

Structured Query Language

Most computer software has not yet reached a point where you can literally ask a database, “what orders have been placed by customers in New Jersey with ship dates in the second quarter?” You must still phrase questions in a restricted syntax that the software can easily parse. You can pose the same question to the demonstration database in the following terms:

```
SELECT * FROM customer, orders
      WHERE customer.customer_num = orders.customer_num
      AND customer.state = 'NJ'
      AND orders.ship_date
      BETWEEN DATE('7/1/93') AND DATE('7/30/93')
```

This question is a sample of Structured Query Language (SQL). It is the language that you use to direct all operations on the database. SQL is composed of statements, each of which begins with one or two keywords that specify a function. SQL includes about 73 statements, from `ALLOCATE DESCRIPTOR` to `WHENEVER`.

All the SQL statements are specified in detail in the *Informix Guide to SQL: Syntax*. Most of the statements are used infrequently—when setting up or tuning a database. People generally use three or four statements to query or update databases.

One statement, `SELECT`, is in almost constant use. It is the only statement that you can use to retrieve data from the database. It is also the most complicated statement, and the next two chapters of this book are devoted to exploring its many uses.

Standard SQL

SQL and the relational model were invented and developed at IBM in the early and middle 1970s. Once IBM proved that it was possible to implement practical relational databases and that SQL was a usable language for manipulating them, other vendors began to provide similar products for non-IBM computers.

For reasons of performance or competitive advantage, or to take advantage of local hardware or software features, each of these SQL implementations differed in small ways from each other and from the IBM version of the language. To ensure that the differences remained small, a standards committee was formed in the early 1980s.

Committee X3H2, sponsored by the American National Standards Institute (ANSI), issued the SQL1 standard in 1986. This standard defines a core set of SQL features and the syntax of statements such as SELECT.

Informix SQL and ANSI SQL

The SQL version that is supported by Informix products is highly compatible with standard SQL (it is also compatible with the IBM version of the language). However, it does contain *extensions* to the standard; that is, extra options or features for certain statements, and looser rules for others. Most of the differences occur in the statements that are not in everyday use. For example, few differences occur in the SELECT statement, which accounts for 90 percent of the SQL use for a typical person.

However, the extensions do exist, and that creates a conflict. Thousands of Informix customers have embedded Informix-style SQL in programs and stored queries. They rely on Informix to keep its language the same. Other customers require the ability to use databases in a way that conforms exactly to the ANSI standard. They rely on Informix to change its language to conform.

Informix resolved the conflict with the following compromise:

- The Informix version of SQL, with its extensions to the standard, is available by default.
- You can ask any Informix SQL language processor to check your use of SQL and post a warning flag whenever you use an Informix extension.

This resolution is fair but makes the SQL documentation more complicated. Wherever a difference exists between Informix and ANSI SQL, the *Informix Guide to SQL: Syntax* describes both versions. Because you probably intend to use only one version, you will have to ignore the version you do not need.

ANSI-Compliant Databases

You can designate a database as ANSI-compliant by using the MODE ANSI keywords when you create it. Within such a database, certain characteristics of the ANSI standard apply. For example, all actions that modify data automatically take place within a transaction, which means that the changes are made in their entirety or not at all. Differences in the behavior of ANSI-compliant databases are noted where appropriate in the *Informix Guide to SQL: Syntax*.

NLS Databases

The Version 6.0 Informix database servers provide Native Language Support (NLS). This allows you to work in various European or Latin American languages other than English and conform to the customs of a specific locale (various money and date formats, foreign characters in data or identifier names, different collation order, and so on). You can activate the NLS functionality by setting the DBNLS environment variable as well as other NLS-specific environment variables.

Chapter 4 of the *Informix Guide to SQL: Reference* describes how to set the NLS environment variables. For additional information on NLS databases, see Chapter 1 in the *Informix Guide to SQL: Reference*.

The Database Software

You access your database through two layers of sophisticated software. The top layer, or *application*, sends commands or queries to the database server. The application calls on the bottom layer, or *database server*, and gets back information. You command both layers using SQL.

The Applications

A *database application*, or simply *application*, is a program that uses the database. It does so by calling on the database server. At its simplest, the application sends SQL commands to the database server, and the database server sends rows of data back to the application. Then the application displays the rows to you, its user.

Alternatively, you command the application to add new data to the database. It incorporates the new data as part of an SQL command to insert a row and passes this command to the database server for execution.

Several types of applications exist. Some allow you to access the database interactively using SQL; others present the stored data in different forms related to its use.

The Database Server

The *database server* is the program that manages the contents of the database as they are stored on disk. The database server knows how tables, rows, and columns are actually organized in physical computer storage. The database server also interprets and executes all SQL commands.

Interactive SQL

To carry out the examples in this book, and to experiment with SQL and database design for yourself, you need a program that lets you execute SQL statements interactively. **DB-Access** and **INFORMIX-SQL** are two such programs. They assist you in composing SQL statements; then they pass your SQL to the database server for execution and display the results to you.

Alternatively, you may use a **Wingz** spreadsheet on a workstation or personal computer. **Wingz**, with its optional **DataLink** extension, can talk to a database server. Using the **Wingz** language, **HyperScript**, you can write SQL statements and have rows of data returned to your spreadsheet for display.

Reports and Forms

After you perfect a query to return precisely the data in which you are interested, you need to format the data for display as a report or form on the terminal screen. **ACE** is the report generator for **INFORMIX-SQL**. You provide to **ACE** a **SELECT** statement that returns the rows of data, and a report specification stating how the pages of the report are to be laid out. **ACE** compiles this information into a program that you can run whenever you want to produce that report.

PERFORM is the module of **INFORMIX-SQL** that generates interactive screen forms. You prepare a form specification that relates display fields on the screen to columns within tables in the database. **PERFORM** compiles this specification into a program that you can run at any time. When run, the form program interrogates the database to display a row or rows of data on the screen in the format you specified. You can arrange the form so that the user can type in sample values and have matching rows returned from the database (the *query-by-example* feature).

For more information on these products, refer to the manuals that come with them.

General Programming

You can write programs that incorporate SQL statements and exchange data with the database server. That is, you can write a program to retrieve data from the database and format it however you choose. You also can write programs that take data from any source in any format, prepare it, and insert it into the database.

The most convenient programming language for this kind of work is **INFORMIX-4GL**, a nonprocedural language designed expressly for writing database applications. However, you can also communicate with an Informix database server from programs that contain embedded SQL written in C, COBOL, Ada, and FORTRAN.

You also can write programs called stored procedures to work with database data and objects. The stored procedures that you write are stored directly in a database within tables. You can then execute a stored procedure from **DB-Access** or an SQL API.

Chapter 5, “Programming with SQL,” and Chapter 6, “Modifying Data Through SQL Programs,” present an overview of how SQL is used in programs.

Applications and Database Servers

Every program that uses data from a database operates the same way. Regardless of whether it is a packaged program such as **INFORMIX-SQL**, a report program compiled by **ACE**, or a program that you wrote using **INFORMIX-4GL** or an SQL API, you find the same two layers in every case:

1. An application that interacts with the user, prepares and formats data, and sets up SQL statements.
2. A database server that manages the database and interprets the SQL.

All the applications converge on the database server, and only the database server manipulates the database files on disk.

Summary

A database contains a collection of related information but differs in a fundamental way from other methods of storing data. The database contains not only the data but also a data model that defines each data item and specifies its meaning with respect to the other items and to the real world.

A database can be used and even modified by several computer users working concurrently. Different users can be given different views of the contents of a database, and their access to those contents can be restricted in several ways.

A database can be crucially important to the success of an organization and can require central administration and monitoring. The **INFORMIX-OnLine Dynamic Server** database server caters to the needs of *essential* applications; both **OnLine** and **INFORMIX-SE** support smaller databases for private or group use.

You manipulate and query a database using SQL, which was pioneered by IBM and standardized by ANSI. You can take advantage of the Informix extensions to the ANSI-defined language, and your Informix tools also make it possible to maintain strict compliance with ANSI standards.

Two layers of software mediate all your work with databases. The bottom layer is always a database server that executes SQL statements and manages the data on disk and in computer memory. The top layer is one of many applications, some from Informix and some written by you or written by other vendors or your colleagues.

Summary

Composing Simple SELECT Statements

Chapter Overview	3
Introducing the SELECT Statement	4
Some Basic Concepts	4
Privileges	5
Relational Operations	5
Selection and Projection	5
Joining	8
The Forms of SELECT	10
Special Data Types	11
Single-Table SELECT Statements	11
Selecting All Columns and Rows	12
Using the Asterisk	12
Reordering the Columns	13
Sorting the Rows	13
Selecting Specific Columns	18
ORDER BY and Native Language Support	25
Selecting Substrings	27
Using the WHERE Clause	28
Creating a Comparison Condition	29
Using Variable Text Searches	37
Using Exact Text Comparisons	37
Using a Single-Character Wildcard	38
MATCHES and Native Language Support	42
Comparing for Special Characters	43
Expressions and Derived Values	46
Arithmetic Expressions	46
Sorting on Derived Columns	52

Using Functions in SELECT Statements	53
Aggregate Functions	53
Time Functions	56
Other Functions and Keywords	63
Using Stored Procedures in SELECT Statements	67
Multiple-Table SELECT Statements	69
Creating a Cartesian Product	69
Creating a Join	71
Equi-Join	71
Natural Join	75
Multiple-Table Join	77
Some Query Shortcuts	79
Using Aliases	79
The INTO TEMP Clause	83
Summary	84

Chapter Overview

SELECT is the most important and the most complex SQL statement. It is used, along with the SQL statements INSERT, UPDATE, and DELETE, to manipulate data. You can use the SELECT statement in the following ways:

- By itself to retrieve data from a database
- As part of an INSERT statement to produce new rows
- As part of an UPDATE statement to update information

The SELECT statement is the primary way to query on information in a database. It is your key to retrieving data in a program, report, screen form, or spreadsheet.

This chapter shows how you can use the SELECT statement to query on and retrieve data in a variety of ways from a relational database. It discusses how to tailor your statements to select columns or rows of information from one or more tables, how to include expressions and functions in SELECT statements, and how to create various join conditions between relational database tables.

This chapter introduces the basic methods for retrieving data from a relational database. More complex uses of SELECT statements, such as subqueries, outer joins, and unions, are discussed in Chapter 3 of this manual. The syntax and usage for the SELECT statement are described in detail in Chapter 1 of the *Informix Guide to SQL: Syntax*.

Most of the examples in this chapter come from the nine tables in the **stores6** demonstration database, which is installed with the software for your Informix SQL API. In the interest of brevity, the examples may show only part of the data that is retrieved for each SELECT statement. See Appendix A in the *Informix Guide to SQL: Reference* for information on the structure and contents of the **stores6** database. For emphasis, keywords are shown in uppercase letters in the examples, although SQL is not case sensitive.

Introducing the SELECT Statement

The SELECT statement is constructed of clauses that let you look at data in a relational database. These clauses let you select columns and rows from one or more database tables or views, specify one or more conditions, order and summarize the data, and put the selected data in a temporary table.

This chapter shows how to use five SELECT statement clauses. You must include these clauses in a SELECT statement in the following order:

1. SELECT clause
2. FROM clause
3. WHERE clause
4. ORDER BY clause
5. INTO TEMP clause

Only the SELECT and FROM clauses are required. These two clauses form the basis for every database query because they specify the tables and columns to be retrieved.

- Add a WHERE clause to select specific rows or create a *join* condition.
- Add an ORDER BY clause to change the order in which data is produced.
- Add an INTO TEMP clause to save the results as a table for further queries.

Two additional SELECT statement clauses, GROUP BY and HAVING, enable you to perform more complex data retrieval. They are introduced in Chapter 3, “Composing Advanced SELECT Statements.” Another clause, INTO, is used to specify the program or host variable to receive data from a SELECT statement in the **INFORMIX-4GL** and SQL APIs. Complete syntax and rules for using the SELECT statement are shown in Chapter 1 of the *Informix Guide to SQL: Syntax*.

Some Basic Concepts

The SELECT statement, unlike INSERT, UPDATE, and DELETE statements, does not modify the data in a database. It is used simply to query on the data. Whereas only one user at a time can modify data, multiple users can query on or *select* the data concurrently. The statements that modify data are discussed in Chapter 4, “Modifying Data.” The INSERT, UPDATE, and DELETE statements appear in Chapter 1 of the *Informix Guide to SQL: Syntax*.

In a relational database, a *column* is a data element that contains a particular type of information that occurs in every row in the table. A *row* is a group of related items of information about a single entity across all columns in a database table.

You can select columns and rows from a database table; from a *system catalog table*, a file that contains information on the database; or from a *view*, which is a virtual table created to contain a customized set of data. System catalog tables are shown in Chapter 2 of the *Informix Guide to SQL: Reference*. Views are discussed in Chapter 1 of the *Informix Guide to SQL: Syntax*.

Privileges

Before you can query on data, you must have the Connect privilege to the database and the Select privilege to the tables in it. These privileges normally are granted to all users as a matter of course. Database access privileges are discussed in Chapter 11, “Granting and Limiting Access to Your Database,” of this manual and in the GRANT and REVOKE statements in Chapter 1 of the *Informix Guide to SQL: Syntax*.

Relational Operations

A *relational operation* involves manipulating one or more tables, or *relations*, to result in another table. The three kinds of relational operation are selection, projection, and join. This chapter includes examples of selection, projection, and simple joining.

Selection and Projection

In relational terminology, *selection* is defined as taking the *horizontal* subset of rows of a single table that satisfies a particular condition. This kind of SELECT statement returns some of the rows and all of the columns in a table. Selection is implemented through the WHERE clause of a SELECT statement, as shown in Query 2-1:

```
SELECT * FROM customer
      WHERE state = 'NJ'
```

Query 2-1

Query Result 2-1 contains the same number of columns as the **customer** table, but only a subset of its rows. Because the data in the selected columns does not fit on one line of the **DB-Access** or **INFORMIX-SQL** Interactive Schema Editor (ISED) screen, the data is displayed vertically instead of horizontally.

```
customer_num  119
fname         Bob
lname        Shorter
company       The Triathletes Club
address1      2405 Kings Highway
address2
city          Cherry Hill
state         NJ
zipcode       08002
phone        609-663-6079

customer_num  122
fname         Cathy
lname        O'Brian
company       The Sporting Life
address1      543 Nassau Street
address2
city          Princeton
state         NJ
zipcode       08540
phone        609-342-0054
```

Query Result 2-1

In relational terminology, *projection* is defined as taking a *vertical* subset from the columns of a single table that retains the unique rows. This kind of SELECT statement returns some of the columns and all of the rows in a table.

Projection is implemented through the select list in the SELECT clause of a SELECT statement, as shown in Query 2-2:

```
SELECT UNIQUE city, state, zipcode
FROM customer
```

Query 2-2

Query Result 2-2 contains the same number of rows as the **customer** table, but it *projects* only a subset of the table's columns:

city	state	zipcode
Bartlesville	OK	74006
Blue Island	NY	60406
Brighton	MA	02135
Cherry Hill	NJ	08002
Denver	CO	80219
Jacksonville	FL	32256
Los Altos	CA	94022
Menlo Park	CA	94025
Mountain View	CA	94040
Mountain View	CA	94063
Oakland	CA	94609
Palo Alto	CA	94303
Palo Alto	CA	94304
Phoenix	AZ	85008
Phoenix	AZ	85016
Princeton	NJ	08540
Redwood City	CA	94026
Redwood City	CA	94062
Redwood City	CA	94063
San Francisco	CA	94117
Sunnyvale	CA	94085
Sunnyvale	CA	94086
Wilmington	DE	19898

Query Result 2-2

The most common kind of SELECT statement uses both selection and projection. A query of this kind, shown in Query 2-3, returns some of the rows and some of the columns in a table:

```
SELECT UNIQUE city, state, zipcode
FROM customer
WHERE state = 'NJ'
```

Query 2-3

Query Result 2-3 contains a subset of the rows and a subset of the columns in the **customer** table:

city	state	zipcode
Cherry Hill	NJ	08002
Princeton	NJ	08540

Query Result 2-3

Joining

A join occurs when two or more tables are connected by one or more columns in common, creating a new table of results. Query 2-4 uses a subset of the **items** and **stock** tables to illustrate the concept of a join, as shown in Figure 2-1.

```
SELECT unique item_num, order_num, stock.stock_num, description
FROM items, stock
WHERE items.stock_num = stock.stock_num
```

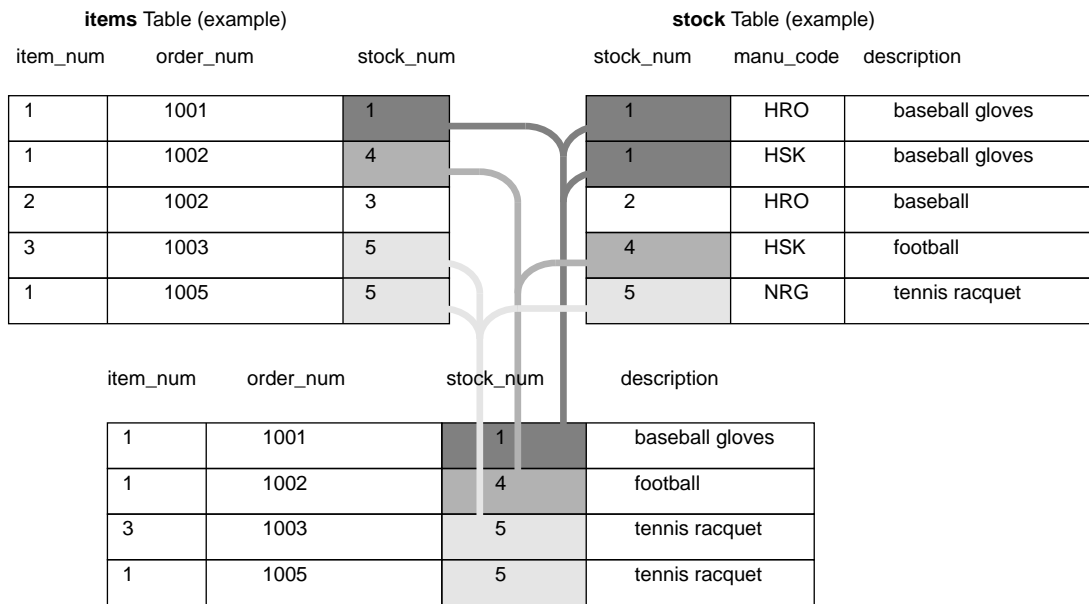


Figure 2-1 Example of a join between two tables

Query 2-4 joins the **customer** and **state** tables:

```
SELECT UNIQUE city, state, zipcode, sname
FROM customer, state
WHERE customer.state = state.code
```

Query 2-4

Query Result 2-4 is composed of specified rows and columns from both the **customer** and **state** tables:

city	state	zipcode	sname
Bartlesville	OK	74006	Oklahoma
Blue Island	NY	60406	New York
Brighton	MA	02135	Massachusetts
Cherry Hill	NJ	08002	New Jersey
Denver	CO	80219	Colorado
Jacksonville	FL	32256	Florida
Los Altos	CA	94022	California
Menlo Park	CA	94025	California
Mountain View	CA	94040	California
Mountain View	CA	94063	California
Oakland	CA	94609	California
Palo Alto	CA	94303	California
Palo Alto	CA	94304	California
Phoenix	AZ	85008	Arizona
Phoenix	AZ	85016	Arizona
Princeton	NJ	08540	New Jersey
Redwood City	CA	94026	California
Redwood City	CA	94062	California
Redwood City	CA	94063	California
San Francisco	CA	94117	California
Sunnyvale	CA	94085	California
Sunnyvale	CA	94086	California
Wilmington	DE	19898	Delaware

Query Result 2-4

The Forms of SELECT

Although the syntax remains the same across all Informix products, the form of a SELECT statement and the location and formatting of the resulting output depends on the application. The examples in this chapter and in Chapter 3, “Composing Advanced SELECT Statements,” display the SELECT statements and their output as they appear when you use the interactive Query-language option in **DB-Access** or **INFORMIX-SQL**.

You also can use SELECT statements to query on data noninteractively through **INFORMIX-SQL ACE** reports, you can embed them in a language such as **INFORMIX-ESQL/C** (where they are treated as executable code), or you can incorporate them in **INFORMIX-4GL** as part of its fourth-generation language.

Special Data Types

The examples in this chapter use the **INFORMIX-OnLine Dynamic Server** database server, which enables database applications to include the data types VARCHAR, TEXT, and BYTE. These data types are not available to applications that run on **INFORMIX-SE**.

With the **DB-Access** or **INFORMIX-SQL** Interactive Editor, when you issue a SELECT statement that includes one of these three data types, the results of the query are displayed differently.

- If you execute a query on a VARCHAR column, the entire VARCHAR value is displayed, just as CHARACTER values are displayed.
- If you select a TEXT column, the contents of the TEXT column are displayed and you can scroll through them.
- If you query on a BYTE column, the words <BYTE value> are displayed instead of the actual value.

Differences specific to VARCHAR, TEXT, and BYTE are noted as appropriate throughout this chapter.

When Native Language Support (NLS) is enabled, you can issue a SELECT statement that queries on NCHAR or NVARCHAR columns instead of their CHAR or VARCHAR counterparts. The NCHAR and NVARCHAR data types allow foreign characters in character data.

For additional information on these and other data types, see Chapter 9, “Implementing Your Data Model,” in this manual, and Chapter 3 in the *Informix Guide to SQL: Reference*.

Single-Table SELECT Statements

You can query a single table in a database in many ways. You can tailor a SELECT statement to perform the following actions:

- Retrieve all or specific columns
- Retrieve all or specific rows
- Perform computations or other functions on the retrieved data
- Order the data in various ways

Selecting All Columns and Rows

The most basic SELECT statement contains only the two required clauses, SELECT and FROM.

Using the Asterisk

Query 2-5a specifies all the columns in the **manufact** table in a *select list*. A select list is a list of the column names or expressions that you want to project from a table.

```
SELECT manu_code, manu_name, lead_time
      FROM manufact
```

Query 2-5a

Query 2-5b uses the *wildcard* * (asterisk), which is shorthand for the select list. The * represents the names of all the columns in the table. You can use the * when you want all the columns, in their defined order.

```
SELECT * FROM manufact
```

Query 2-5b

Query 2-5a and Query 2-5b are equivalent and display the same results; that is, a list of every column and row in the **manufact** table. Query Result 2-5 shows the results as they would appear on a DB-Access or INFORMIX-SQL Interactive Schema Editor (ISED) screen.

manu_code	manu_name	lead_time
SMT	Smith	3
ANZ	Anza	5
NRG	Norge	7
HSK	Husky	5
HRO	Hero	4
SHM	Shimara	30
KAR	Karsten	21
NKL	Nikolus	8
PRC	ProCycle	9

Query Result 2-5

Reordering the Columns

Query 2-6 shows how you can change the order in which the columns are listed by changing their order in your select list:

```
SELECT manu_name, manu_code, lead_time
FROM manufact
```

Query 2-6

Query Result 2-6 includes the same columns as the previous one, but because the columns are specified in a different order, the display is also different:

manu_name	manu_code	lead_time
Smith	SMT	3
Anza	ANZ	5
Norge	NRG	7
Husky	HSK	5
Hero	HRO	4
Shimara	SHM	30
Karsten	KAR	21
Nikolus	NKL	8
ProCycle	PRC	9

Query Result 2-6

Sorting the Rows

You can direct the system to sort the data in a specific order by adding an **ORDER BY** clause to your **SELECT** statement. The columns you want to use in the **ORDER BY** clause must be included in the select list either explicitly or implicitly.

An *explicit* select list, shown in Query 2-7a, includes all the column names that you want to retrieve:

```
SELECT manu_code, manu_name, lead_time
FROM manufact
ORDER BY lead_time
```

Query 2-7a

An *implicit* select list uses the * symbol, as shown in Query 2-7b:

```
SELECT * FROM manufact
      ORDER BY lead_time
```

Query 2-7b

Query 2-7a and Query 2-7b produce the same display. Query Result 2-7 shows a list of every column and row in the **manufact** table, in order of **lead_time**:

manu_code	manu_name	lead_time
SMT	Smith	3
HRO	Hero	4
HSK	Husky	5
ANZ	Anza	5
NRG	Norge	7
NKL	Nikolus	8
PRC	ProCycle	9
KAR	Karsten	21
SHM	Shimara	30

Query Result 2-7

Ascending Order

The retrieved data is sorted and displayed, by default, in *ascending* order. Ascending order is uppercase A to lowercase z for CHARACTER data types, and lowest to highest value for numeric data types. DATE and DATETIME type data are sorted from earliest to latest, and INTERVAL data is ordered from shortest to longest span of time.

Descending Order

Descending order is the opposite of ascending order, from lowercase z to uppercase A for character types and highest to lowest for numeric data types. DATE and DATETIME data are sorted from latest to earliest, and INTERVAL data is ordered from longest to shortest span of time. Query 2-8 shows an example of descending order:

```
SELECT * FROM manufact
      ORDER BY lead_time DESC
```

Query 2-8

The keyword DESC following a column name causes the retrieved data to be sorted in *descending* order, as shown in Query Result 2-8:

manu_code	manu_name	lead_time
SHM	Shimara	30
KAR	Karsten	21
PRC	ProCycle	9
NKL	Nikolus	8
NRG	Norge	7
HSK	Husky	5
ANZ	Anza	5
HRO	Hero	4
SMT	Smith	3

Query Result 2-8

You can specify any column (except TEXT or BYTE) in the ORDER BY clause, and the database server sorts the data based on the values in that column.

Sorting on Multiple Columns

You can also ORDER BY two or more columns, creating a *nested sort*. The default is still ascending, and the column listed first in the ORDER BY clause takes precedence.

The following two queries and corresponding query results show nested sorts. The order in which selected data is displayed is modified by changing the order of the two columns named in the ORDER BY clause.

```
SELECT * FROM stock
      ORDER BY manu_code, unit_price
```

Query 2-9

In Query Result 2-9, the **manu_code** column data appears in alphabetical order and, within each set of rows with the same **manu_code** (for example, ANZ, HRO), the **unit_price** is listed in ascending order:

stock_num	manu_code	description	unit_price	unit	unit_descr
5	ANZ	tennis racquet	\$19.80	each	each
9	ANZ	volleyball net	\$20.00	each	each
6	ANZ	tennis ball	\$48.00	case	24 cans/case
313	ANZ	swim cap	\$60.00	box	12/box
201	ANZ	golf shoes	\$75.00	each	each
310	ANZ	kick board	\$84.00	case	12/case
301	ANZ	running shoes	\$95.00	each	each
304	ANZ	watch	\$170.00	box	10/box
110	ANZ	helmet	\$244.00	case	4/case
205	ANZ	3 golf balls	\$312.00	case	24/case
8	ANZ	volleyball	\$840.00	case	24/case
302	HRO	ice pack	\$4.50	each	each
309	HRO	ear drops	\$40.00	case	20/case
.					
.					
.					
113	SHM	18-spd, assmbld	\$685.90	each	each
5	SMT	tennis racquet	\$25.00	each	each
6	SMT	tennis ball	\$36.00	case	24 cans/case
1	SMT	baseball gloves	\$450.00	case	10 gloves/case

Query Result 2-9

Query 2-10 shows the reversed order of the columns in the ORDER BY clause:

```
SELECT * FROM stock
      ORDER BY unit_price, manu_code
```

Query 2-10

In Query Result 2-10, the data appears in ascending order of **unit_price** and, where two or more rows have the same **unit_price** (for example, \$20.00, \$48.00, \$312.00), the **manu_code** is in alphabetical order:

stock_num	manu_code	description	unit_price	unit	unit_descr
302	HRO	ice pack	\$4.50	each	each
302	KAR	ice pack	\$5.00	each	each
5	ANZ	tennis racquet	\$19.80	each	each
9	ANZ	volleyball net	\$20.00	each	each
103	PRC	frnt derailleur	\$20.00	each	each
106	PRC	bicycle stem	\$23.00	each	each
5	SMT	tennis racquet	\$25.00	each	each
.					
.					
.					
301	HRO	running shoes	\$42.50	each	each
204	KAR	putter	\$45.00	each	each
108	SHM	crankset	\$45.00	each	each
6	ANZ	tennis ball	\$48.00	case	24 cans/case
305	HRO	first-aid kit	\$48.00	case	4/case
303	PRC	socks	\$48.00	box	24 pairs/box
311	SHM	water gloves	\$48.00	box	4 pairs/box
.					
.					
.					
110	HSK	helmet	\$308.00	case	4/case
205	ANZ	3 golf balls	\$312.00	case	24/case
205	HRO	3 golf balls	\$312.00	case	24/case
205	NKL	3 golf balls	\$312.00	case	24/case
1	SMT	baseball gloves	\$450.00	case	10 gloves/case
4	HRO	football	\$480.00	case	24/case
102	PRC	bicycle brakes	\$480.00	case	4 sets/case
111	SHM	10-spd, assmbld	\$499.99	each	each
112	SHM	12-spd, assmbld	\$549.00	each	each
7	HRO	basketball	\$600.00	case	24/case
203	NKL	irons/wedge	\$670.00	case	2 sets/case
113	SHM	18-spd, assmbld	\$685.90	each	each
1	HSK	baseball gloves	\$800.00	case	10 gloves/case
8	ANZ	volleyball	\$840.00	case	24/case
4	HSK	football	\$960.00	case	24/case

Query Result 2-10

The order of the columns in the ORDER BY clause is important, and so is the position of the DESC keyword. Although the statements in Query 2-11 contain the same components in the ORDER BY clause, each produces a different result (not shown):

```
SELECT * FROM stock
  ORDER BY manu_code, unit_price DESC

SELECT * FROM stock
  ORDER BY unit_price, manu_code DESC

SELECT * FROM stock
  ORDER BY manu_code DESC, unit_price

SELECT * FROM stock
  ORDER BY unit_price DESC, manu_code
```

Query 2-11

Selecting Specific Columns

The previous section showed how to select and order all data from a table. However, often all you want to see is the data in one or more specific columns. Again, the formula is to use the SELECT and FROM clauses, specify the columns and table, and perhaps order the data in ascending or descending order with an ORDER BY clause.

If you want to find all the customer numbers in the **orders** table, you would use a statement such as the one in Query 2-12:

```
SELECT customer_num FROM orders
```

Query 2-12

Query Result 2-12 shows how the statement simply selects all data in the **customer_num** column in the **orders** table and lists the customer numbers on all the orders, including duplicates:

```
customer_num
101
104
104
104
104
106
106
110
110
111
112
115
116
117
117
119
120
121
122
123
124
126
127
```

Query Result 2-12

The output includes several duplicates because some customers have placed more than one order. Sometimes you want to see duplicate rows in a projection. At other times, you want to see only the distinct values, not how often each value appears.

You can cause duplicate rows to be suppressed by including the keyword **DISTINCT** or its synonym **UNIQUE** at the start of the select list, as shown in Query 2-13:

```
SELECT DISTINCT customer_num FROM orders

SELECT UNIQUE customer_num FROM orders
```

Query 2-13

Query 2-13 limits the display to show each customer number in the **orders** table only once, producing a more readable list, as shown in Query Result 2-13:

```
customer_num
          101
          104
          106
          110
          111
          112
          115
          116
          117
          119
          120
          121
          122
          123
          124
          126
          127
```

Query Result 2-13

Suppose you are handling a customer call, and you want to locate purchase order number DM354331. You decide to list all the purchase order numbers in the **orders** table, using a statement such as the one in Query 2-14:

```
SELECT po_num FROM orders
```

Query 2-14

Query Result 2-14 shows how the statement retrieves data in the **po_num** column in the **orders** table:

```
po_num  
  
B77836  
9270  
B77890  
8006  
2865  
Q13557  
278693  
LZ230  
4745  
429Q  
B77897  
278701  
B77930  
8052  
MA003  
PC6782  
DM354331  
S22942  
Z55709  
W2286  
C3288  
W9925  
KF2961
```

Query Result 2-14

However, the list is not in a very useful order. You can add an ORDER BY clause to sort the column data in ascending order and make it easier to find that particular **po_num**, as shown in Query 2-15:

```
SELECT po_num FROM orders
      ORDER BY po_num
```

Query 2-15

```
po_num
278693
278701
2865
429Q
4745
8006
8052
9270
B77836
B77890
B77897
B77930
C3288
DM354331
KF2961
LZ230
MA003
PC6782
Q13557
S22942
W2286
W9925
Z55709
```

Query Result 2-15

To select multiple columns from a table, list them in the select list in the SELECT clause. Query 2-16 shows the order in which the columns are selected is the order in which they are produced, from left to right:

```
SELECT paid_date, ship_date, order_date,
       customer_num, order_num, po_num
FROM orders
ORDER BY paid_date, order_date, customer_num
```

Query 2-16

As shown in “Sorting on Multiple Columns” on page 2-15, you can use the ORDER BY clause to sort the data in ascending or descending order and perform nested sorts. Query Result 2-16 shows descending order:

paid_date	ship_date	order_date	customer_num	order_num	po_num
	05/30/1993	05/22/1993	106	1004	8006
		05/30/1993	112	1006	Q13557
	06/05/1993	05/31/1993	117	1007	278693
	06/29/1993	06/18/1993	117	1012	278701
	07/12/1993	06/29/1993	119	1016	PC6782
	07/13/1993	07/09/1993	120	1017	DM354331
06/03/1993	05/26/1993	05/21/1993	101	1002	9270
06/14/1993	05/23/1993	05/22/1993	104	1003	B77890
06/21/1993	06/09/1993	05/24/1993	116	1005	2865
07/10/1993	07/03/1993	06/25/1993	106	1014	8052
07/21/1993	07/06/1993	06/07/1993	110	1008	LZ230
07/22/1993	06/01/1993	05/20/1993	104	1001	B77836
07/31/1993	07/10/1993	06/22/1993	104	1013	B77930
08/06/1993	07/13/1993	07/10/1993	121	1018	S22942
08/06/1993	07/16/1993	07/11/1993	122	1019	Z55709
08/21/1993	06/21/1993	06/14/1993	111	1009	4745
08/22/1993	06/29/1993	06/17/1993	115	1010	429Q
08/22/1993	07/25/1993	07/23/1993	124	1021	C3288
08/22/1993	07/30/1993	07/24/1993	127	1023	KF2961
08/29/1993	07/03/1993	06/18/1993	104	1011	B77897
08/31/1993	07/16/1993	06/27/1993	110	1015	MA003
09/02/1993	07/30/1993	07/24/1993	126	1022	W9925
09/20/1993	07/16/1993	07/11/1993	123	1020	W2286

Query Result 2-16

When you use SELECT and ORDER BY on several columns in a table, you might find it helpful to use integers to refer to the position of the columns in the ORDER BY clause. The statements in Query 2-17 retrieve and display the same data, shown in Query Result 2-17.

```
SELECT customer_num, order_num, po_num, order_date
FROM orders
ORDER BY 4, 1

SELECT customer_num, order_num, po_num, order_date
FROM orders
ORDER BY order_date, customer_num
```

Query 2-17

customer_num	order_num	po_num	order_date
104	1001	B77836	05/20/1993
101	1002	9270	05/21/1993
104	1003	B77890	05/22/1993
106	1004	8006	05/22/1993
116	1005	2865	05/24/1993
112	1006	Q13557	05/30/1993
117	1007	278693	05/31/1993
110	1008	LZ230	06/07/1993
111	1009	4745	06/14/1993
115	1010	429Q	06/17/1993
104	1011	B77897	06/18/1993
117	1012	278701	06/18/1993
104	1013	B77930	06/22/1993
106	1014	8052	06/25/1993
110	1015	MA003	06/27/1993
119	1016	PC6782	06/29/1993
120	1017	DM354331	07/09/1993
121	1018	S22942	07/10/1993
122	1019	Z55709	07/11/1993
123	1020	W2286	07/11/1993
124	1021	C3288	07/23/1993
126	1022	W9925	07/24/1993
127	1023	KF2961	07/24/1993

Query Result 2-17

You can include the DESC keyword in the ORDER BY clause when you assign integers to column names, as shown in Query 2-18:

```
SELECT customer_num, order_num, po_num, order_date
      FROM orders
      ORDER BY 4 DESC, 1
```

Query 2-18

In this case, data is first sorted in descending order by **order_date** and in ascending order by **customer_num**.

ORDER BY and Native Language Support

You can set the DBNLS environment variable to activate Native Language Support (NLS) functionality. You can then specify a language environment (locale) by setting the LANG environment variable and can set the LC_COLLATE X/Open category to further define the collation order.

Query 2-18 uses a SELECT statement with an ORDER BY clause on a column containing data of type NCHAR (the NLS equivalent of CHAR). This NLS table, **abonnés**, is provided separately from the demonstration database; this SELECT statement example is provided as the **sel_nls1.sql** command file with NLS-enabled **DB-Access**.

```
SELECT numéro,nom,prénom
      FROM abonnés
      ORDER BY nom;
```

Query 2-19

The collation order for the results of this query may vary, depending on whether NLS is activated and the settings for the LANG and LC_COLLATE environment variables, as well as how uppercase versus lowercase letters are handled by the particular system. Query Result 2-19a and Query Result 2-19b show two sample sets of output:

numéro	nom	prénom
13612	Azevedo	Edouardo Freire
13606	Dupré	Michèle Françoise
13607	Hammer	Gerhard
13602	Hämmerle	Greta
13604	LaForêt	Jean-Noël
13610	LeMaître	Héloïse
13613	Llanero	Gloria Dolores
13603	Montaña	José Antonio
13611	Oatfield	Emily
13609	Tiramisù	Paolo Alfredo
13600	da Sousa	João Lourenço Antunes
13615	di Girolamo	Giuseppe
13601	Ålesund	Sverre
13608	Étaix	Émile
13605	Ötker	Hans-Jürgen
13614	Øverst	Per-Anders

Query Result 2-19a

Query Result 2-19a follows the standard ASCII sorting order, which ranks capital letters before lowercase letters and moves the names that start with a capitalized foreign character (Ålesund, Étaix, Ötker, and Øverst) to the end of the list.

numéro	nom	prénom
13601	Ålesund	Sverre
13612	Azevedo	Edouardo Freire
13600	da Sousa	João Lourenço Antunes
13615	di Girolamo	Giuseppe
13606	Dupré	Michèle Françoise
13608	Étaix	Émile
13607	Hammer	Gerhard
13602	Hämmerle	Greta
13604	LaForêt	Jean-Noël
13610	LeMaître	Héloïse
13613	Llanero	Gloria Dolores
13603	Montaña	José Antonio
13611	Oatfield	Emily
13605	Ötker	Hans-Jürgen
13614	Øverst	Per-Anders
13609	Tiramisù	Paolo Alfredo

Query Result 2-19b

Query Result 2-19b indicates that with the appropriate NLS environment variable settings, names starting with foreign characters (Ålesund, Étaix, Ötker, and Øverst) are collated differently than they are in the standard ASCII order so they are integrated in the list. It does not distinguish between uppercase and lowercase letters.

Selecting Substrings

You can select part of the value of a CHARACTER column by including a *substring* in the select list. Suppose your marketing department is planning a mailing to your customers and wants a rough idea of their geographical distribution based on zip codes. You could write a query similar to Query 2-20:

```
SELECT zipcode[1,3], customer_num
FROM customer
ORDER BY zipcode
```

Query 2-20

Query 2-20 uses a substring to select the first three characters of the **zipcode** column (which identify the state) and the full **customer_num**, and lists them in ascending order by zip code, as shown in Query Result 2-20:

zipcode	customer_num
021	125
080	119
085	122
198	121
322	123
604	127
740	124
802	126
850	128
850	120
940	105
940	112
940	113
940	115
940	104
940	116
940	110
940	114
940	106
940	108
940	117
940	111
940	101
940	109
941	102
943	103
943	107
946	118

Query Result 2-20

Using the WHERE Clause

Perhaps you want to see only those orders placed by a particular customer or the calls entered by a particular customer service representative. You can add a WHERE clause to a SELECT statement to retrieve these specific rows from a table.

You can use the WHERE clause to set up a *comparison condition* or a *join condition*. This section demonstrates only the first use. Join conditions are described in a later section and in the next chapter.

The set of rows returned by a SELECT statement is the *active set* for that statement. A *singleton* SELECT statement returns a single row. In **INFORMIX-4GL** or an SQL API, the retrieval of multiple rows requires the use of a *cursor*. See Chapter 5, “Programming with SQL,” and Chapter 6, “Modifying Data Through SQL Programs.”

Creating a Comparison Condition

The WHERE clause of a SELECT statement specifies the rows you want to see. A comparison condition employs specific *keywords* and *operators* to define the search criteria.

For example, you might use one of the keywords BETWEEN, IN, LIKE, or MATCHES to test for equality, or the keywords IS NULL to test for null values. You can combine the keyword NOT with any of these keywords to specify the opposite condition.

Figure 2-2 lists the *relational operators* you can use in a WHERE clause in place of a keyword to test for equality.

Operator	Operation
=	equals
!= or <>	does not equal
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Figure 2-2 *Relational operators that test for equality*

For CHAR expressions, “greater than” means *after* in ASCII collating order, where lowercase letters are after uppercase letters, and both are after numerals. See the ASCII Character Set chart in Chapter 1 of the *Informix Guide to SQL: Syntax*. For DATE and DATETIME expressions, “greater than” means *later in time*, and for INTERVAL expressions, it means *of longer duration*. You cannot use TEXT or BYTE columns in string expressions, except when you test for null values.

You can use the preceding keywords and operators in a WHERE clause to create comparison condition queries that perform the following actions:

- Include values
- Exclude values
- Find a range of values

- Find a subset of values
- Identify null values

You can also use the preceding keywords and operators in a WHERE clause to create comparison condition queries, which perform variable text searches using the following criteria:

- Exact text comparison
- Single-character wildcards
- Restricted single-character wildcards
- Variable-length wildcards
- Subscripting

The following section contains examples that illustrate these types of queries.

Including Rows

Use the relational operator = to include rows in a WHERE clause, as shown in Query 2-21:

```
SELECT customer_num, call_code, call_dtime, res_dtime
FROM cust_calls
WHERE user_id = 'maryj'
```

Query 2-21

Query 2-21 returns the set of rows shown in Query Result 2-21:

customer_num	call_code	call_dtime	res_dtime
106	D	1993-06-12 08:20	1993-06-12 08:25
121	O	1993-07-10 14:05	1993-07-10 14:06
127	I	1993-07-31 14:30	

Query Result 2-21

Excluding Rows

Use the relational operators != or <> to exclude rows in a WHERE clause.

Query 2-22 assumes that you are selecting from an ANSI-compliant database; the statements specify the *owner* or login name of the creator of the **customer** table. This qualifier is not required when the creator of the table is the current

user, or when the database is not ANSI-compliant, although it is not incorrect to include it in either case. For a complete discussion of *owner naming*, see Chapter 1 in the *Informix Guide to SQL: Syntax*.

```
SELECT customer_num, company, city, state
  FROM odin.customer
 WHERE state != 'CA'

SELECT customer_num, company, city, state
  FROM odin.customer
 WHERE state <> 'CA'
```

Query 2-22

Both statements in Query 2-22 exclude values by specifying that, in the **customer** table owned by the user **odin**, the value in the **state** column should not be equal to CA .

customer_num	company	city	state
119	The Triathletes Club	Cherry Hill	NJ
120	Century Pro Shop	Phoenix	AZ
121	City Sports	Wilmington	DE
122	The Sporting Life	Princeton	NJ
123	Bay Sports	Jacksonville	FL
124	Putnum's Putters	Bartlesville	OK
125	Total Fitness Sports	Brighton	MA
126	Neelie's Discount Sp	Denver	CO
127	Big Blue Bike Shop	Blue Island	NY
128	Phoenix College	Phoenix	AZ

Query Result 2-22

Specifying Rows

Query 2-23 shows two ways to specify rows in a WHERE clause:

```
SELECT catalog_num, stock_num, manu_code, cat_advert
  FROM catalog
 WHERE catalog_num BETWEEN 10005 AND 10008

SELECT catalog_num, stock_num, manu_code, cat_advert
  FROM catalog
 WHERE catalog_num >= 10005 AND catalog_num <= 10008
```

Query 2-23

Each statement in Query 2-23 specifies a range for **catalog_num** from 10005 through 10008, inclusive. The first statement uses keywords, and the second uses relational operators to retrieve the rows shown in Query Result 2-23:

```
catalog_num 10005
stock_num   3
manu_code    HSK
cat_advert   High-Technology Design Expands the Sweet Spot

catalog_num 10006
stock_num   3
manu_code    SHM
cat_advert   Durable Aluminum for High School and Collegia
te           Athletes

catalog_num 10007
stock_num   4
manu_code    HSK
cat_advert   Quality Pigskin with Joe Namath Signature

catalog_num 10008
stock_num   4
manu_code    HRO
cat_advert   Highest Quality Football for High School
              and Collegiate Competitions
```

Query Result 2-23

Note that although the **catalog** table includes a column with the BYTE data type, that column is not included in this SELECT statement because the output would show only the words <BYTE value> by the column name. You can display TEXT and BYTE values by using the PROGRAM attribute when using forms in INFORMIX-SQL or INFORMIX-4GL or by writing a 4GL or an SQL API to do so.

Excluding a Range of Rows

In Query 2-24, by using the keywords NOT BETWEEN, the condition excludes rows that have the character range 94000 through 94999 in the zipcode column, as shown in Query Result 2-24:

```
SELECT fname, lname, company, city, state
FROM customer
WHERE zipcode NOT BETWEEN '94000' AND '94999'
ORDER BY state
```

Query 2-24

fname	lname	company	city	state
Fred	Jewell	Century* Pro Shop	Phoenix	AZ
Frank	Lessor	Phoenix University	Phoenix	AZ
Eileen	Neelie	Neelie's Discount Sp	Denver	CO
Jason	Wallack	City Sports	Wilmington	DE
Marvin	Hanlon	Bay Sports	Jacksonville	FL
James	Henry	Total Fitness Sports	Brighton	MA
Bob	Shorter	The Triathletes Club	Cherry Hill	NJ
Cathy	O'Brian	The Sporting Life	Princeton	NJ
Kim	Satifer	Big Blue Bike Shop	Blue Island	NY
Chris	Putnum	Putnum's Putters	Bartlesville	OK

Query Result 2-24

Using a WHERE Clause to Find a Subset of Values

As shown in “Excluding Rows” on page 2-30, Query 2-25 also assumes the use of an ANSI-compliant database. The owner qualifier is in quotation marks to preserve the case sensitivity of the literal string.

```
SELECT lname, city, state, phone
FROM 'Aleta'.customer
WHERE state = 'AZ' OR state = 'NJ'
ORDER BY lname
```

```
SELECT lname, city, state, phone
FROM 'Aleta'.customer
WHERE state IN ('AZ', 'NJ')
ORDER BY lname
```

Query 2-25

Each statement in Query 2-25 retrieves rows that include the subset of AZ or NJ in the **state** column of the **Aleta.customer** table, as shown in Query Result 2-25:

lname	city	state	phone
Jewell	Phoenix	AZ	602-265-8754
Lessor	Phoenix	AZ	602-533-1817
O'Brian	Princeton	NJ	609-342-0054
Shorter	Cherry Hill	NJ	609-663-6079

Query Result 2-25

Note that you cannot test a TEXT or BYTE column with the IN keyword.

In Query 2-26, an example of a query on an ANSI-compliant database, no quotation marks exist around the table owner name. Whereas the two statements in Query 2-25 searched the **Aleta.customer** table, Query 2-26 searches the table **ALETA.customer**, which is a different table, due to how ANSI-compliant databases look at owner names.

```
SELECT lname, city, state, phone
FROM Aleta.customer
WHERE state NOT IN ('AZ', 'NJ')
ORDER BY state
```

Query 2-26

In Query 2-26, by adding the keyword **NOT IN**, the subset changes to exclude the subsets **AZ** and **NJ** in the **state** column. Query Result 2-26 shows the results in order of **state**:

lname	city	state	phone
Pauli	Sunnyvale	CA	408-789-8075
Sadler	San Francisco	CA	415-822-1289
Currie	Palo Alto	CA	415-328-4543
Higgins	Redwood City	CA	415-368-1100
Vector	Los Altos	CA	415-776-3249
Watson	Mountain View	CA	415-389-8789
Ream	Palo Alto	CA	415-356-9876
Quinn	Redwood City	CA	415-544-8729
Miller	Sunnyvale	CA	408-723-8789
Jaeger	Redwood City	CA	415-743-3611
Keyes	Sunnyvale	CA	408-277-7245
Lawson	Los Altos	CA	415-887-7235
Beatty	Menlo Park	CA	415-356-9982
Albertson	Redwood City	CA	415-886-6677
Grant	Menlo Park	CA	415-356-1123
Parmelee	Mountain View	CA	415-534-8822
Sipes	Redwood City	CA	415-245-4578
Baxter	Oakland	CA	415-655-0011
Neelie	Denver	CO	303-936-7731
Wallack	Wilmington	DE	302-366-7511
Hanlon	Jacksonville	FL	904-823-4239
Henry	Brighton	MA	617-232-4159
Satifer	Blue Island	NY	312-944-5691
Putnum	Bartlesville	OK	918-355-2074

Query Result 2-26

Identifying Null Values

Use the **IS NULL** or **IS NOT NULL** option to check for null values. A null represents either the absence of data or a value that is unknown or not applicable. A null value is not the same as a zero or a blank.

Query 2-27 returns all rows that have a null **paid_date**:

```
SELECT order_num, customer_num, po_num, ship_date
FROM orders
WHERE paid_date IS NULL
ORDER BY customer_num
```

Query 2-27

order_num	customer_num	po_num	ship_date
1004	106	8006	05/30/1993
1006	112	Q13557	
1007	117	278693	06/05/1993
1012	117	278701	06/29/1993
1016	119	PC6782	07/12/1993
1017	120	DM354331	07/13/1993

Query Result 2-27

Forming Compound Conditions

You can connect two or more comparison conditions, or *Boolean* expressions, by using the *logical operators* AND, OR, and NOT. A Boolean expression evaluates as true or false or, if null values are involved, as unknown. You can use TEXT or BYTE objects in a Boolean expression only when you test for a null value.

In Query 2-28, the operator AND combines two comparison expressions in the WHERE clause:

```
SELECT order_num, customer_num, po_num, ship_date
FROM orders
WHERE paid_date IS NULL
      AND ship_date IS NOT NULL
ORDER BY customer_num
```

Query 2-28

Query 2-28 returns all rows that have a null **paid_date** and the ones that do not also have a null **ship_date**, as shown in Query Result 2-28:

order_num	customer_num	po_num	ship_date
1004	106	8006	05/30/1993
1007	117	278693	06/05/1993
1012	117	278701	06/29/1993
1016	119	PC6782	07/12/1993

Query Result 2-28

Using Variable Text Searches

You can use the keywords LIKE and MATCHES for *variable text* queries based on substring searches of CHARACTER fields. Include the keyword NOT to indicate the opposite condition. The keyword LIKE is the ANSI standard, whereas MATCHES is an Informix extension.

Variable text search strings can include the wildcards listed in Figure 2-3 with LIKE or MATCHES.

Symbol	Meaning
LIKE	
%	evaluates to zero or more characters
_	evaluates to a single character
\	escapes special significance of next character
MATCHES	
*	evaluates to zero or more characters
?	evaluates to a single character (except null)
[]	evaluates to single character or range of values
\	escapes special significance of next character

Figure 2-3 Wildcards used with LIKE and MATCHES

You cannot test a TEXT or BYTE column with LIKE or MATCHES.

Using Exact Text Comparisons

The following examples include a WHERE clause that searches for exact text comparisons by using the keyword LIKE or MATCHES or the relational operator =. Unlike earlier examples, these examples illustrate how to query on an *external* table in an ANSI-compliant database.

An external table is a table that is not in the current database. You can access only external tables that are part of an ANSI-compliant database.

Whereas the database used previously in this chapter was the demonstration database called **stores6**, the FROM clause in the following examples specifies the **manatee** table, created by the owner **bubba**, which resides in an ANSI-compliant database named **syzygy**. For more information on defining external tables, see Chapter 1 in the *Informix Guide to SQL: Syntax*.

Each statement in Query 2-29 retrieves all the rows that have the single word **helmet** in the **description** column:

```
SELECT * FROM syzygy:bubba.manatee
  WHERE description = 'helmet'
  ORDER BY mfg_code

SELECT * FROM syzygy:bubba.manatee
  WHERE description LIKE 'helmet'
  ORDER BY mfg_code

SELECT * FROM syzygy:bubba.manatee
  WHERE description MATCHES 'helmet'
  ORDER BY mfg_code
```

Query 2-29

stock_no	mfg_code	description	unit_price	unit	unit_type
991	ANT	helmet	\$222.00	case	4/case
991	BKE	helmet	\$269.00	case	4/case
991	JSK	helmet	\$311.00	each	4/case
991	PRM	helmet	\$234.00	case	4/case
991	SHR	helmet	\$245.00	case	4/case

Query Result 2-29

Using a Single-Character Wildcard

The statements in Query 2-30 illustrate the use of a single-character wildcard in a WHERE clause. Further, they demonstrate a query on an external table. The **stock** table is in the external database **sloth**, which, beside being outside the current **stores6** database, is on a separate database server called **meerkat**.

For details on external tables, external databases, and networks, see Chapter 12, “Understanding Informix Networking,” in this manual and Chapter 1 in the *Informix Guide to SQL: Syntax*.

```
SELECT * FROM sloth@meerkat:stock
  WHERE manu_code LIKE '_R_'
        AND unit_price >= 100
        ORDER BY description, unit_price

SELECT * FROM sloth@meerkat:stock
  WHERE manu_code MATCHES '?R?'
        AND unit_price >= 100
        ORDER BY description, unit_price
```

Query 2-30

Each statement in Query 2-30 retrieves only those rows for which the middle letter of the **manu_code** is R, as shown in Query Result 2-30:

stock_num	manu_code	description	unit_price	unit	unit_descr
205	HRO	3 golf balls	\$312.00	case	24/case
2	HRO	baseball	\$126.00	case	24/case
1	HRO	baseball gloves	\$250.00	case	10 gloves/case
7	HRO	basketball	\$600.00	case	24/case
102	PRC	bicycle brakes	\$480.00	case	4 sets/case
114	PRC	bicycle gloves	\$120.00	case	10 pairs/case
4	HRO	football	\$480.00	case	24/case
110	PRC	helmet	\$236.00	case	4/case
110	HRO	helmet	\$260.00	case	4/case
307	PRC	infant jogger	\$250.00	each	each
306	PRC	tandem adapter	\$160.00	each	each
308	PRC	twin jogger	\$280.00	each	each
304	HRO	watch	\$280.00	box	10/box

Query Result 2-30

The comparison `'_R_'` (for `LIKE`) or `'?R?'` (for `MATCHES`) specifies, from left to right, the following items:

- Any single character
- The letter R
- Any single character

WHERE Clause with Restricted Single-Character Wildcard

Query 2-31 selects only those rows where the manu_code begins with A through H and returns the rows shown in Query Result 2-31. The class test '[A-H]' specifies any single letter from A through H, inclusive. No equivalent wildcard symbol exists for the LIKE keyword.

```
SELECT * FROM stock
WHERE manu_code MATCHES '[A-H]*'
ORDER BY description, manu_code, unit_price
```

Query 2-31

stock_num	manu_code	description	unit_price	unit	unit_descr
205	ANZ	3 golf balls	\$312.00	case	24/case
205	HRO	3 golf balls	\$312.00	case	24/case
2	HRO	baseball	\$126.00	case	24/case
3	HSK	baseball bat	\$240.00	case	12/case
1	HRO	baseball gloves	\$250.00	case	10 gloves/case
1	HSK	baseball gloves	\$800.00	case	10 gloves/case
7	HRO	basketball	\$600.00	case	24/case
.					
.					
.					
110	ANZ	helmet	\$244.00	case	4/case
110	HRO	helmet	\$260.00	case	4/case
110	HSK	helmet	\$308.00	case	4/case
.					
.					
.					
301	ANZ	running shoes	\$95.00	each	each
301	HRO	running shoes	\$42.50	each	each
313	ANZ	swim cap	\$60.00	box	12/box
6	ANZ	tennis ball	\$48.00	case	24 cans/case
5	ANZ	tennis racquet	\$19.80	each	each
8	ANZ	volleyball	\$840.00	case	24/case
9	ANZ	volleyball net	\$20.00	each	each
304	ANZ	watch	\$170.00	box	10/box
304	HRO	watch	\$280.00	box	10/box

Query Result 2-31

WHERE Clause with Variable-Length Wildcard

The statements in Query 2-32 use a wildcard at the end of a string to retrieve all the rows where the **description** begins with the characters **bicycle**:

```
SELECT * FROM stock
  WHERE description LIKE 'bicycle%'
     ORDER BY description, manu_code

SELECT * FROM stock
  WHERE description MATCHES 'bicycle*'
     ORDER BY description, manu_code
```

Query 2-32

Either statement returns the rows found in Query Result 2-32:

stock_num	manu_code	description	unit_price	unit	unit_descr
102	PRC	bicycle brakes	\$480.00	case 4	sets/case
102	SHM	bicycle brakes	\$220.00	case 4	sets/case
114	PRC	bicycle gloves	\$120.00	case 10	pairs/case
107	PRC	bicycle saddle	\$70.00	pair	pair
106	PRC	bicycle stem	\$23.00	each	each
101	PRC	bicycle tires	\$88.00	box 4	/box
101	SHM	bicycle tires	\$68.00	box 4	/box
105	PRC	bicycle wheels	\$53.00	pair	pair
105	SHM	bicycle wheels	\$80.00	pair	pair

Query Result 2-32

The comparison **'bicycle%'** or **'bicycle*'** specifies the characters **bicycle** followed by any sequence of zero or more characters. It matches **bicycle stem** with **stem** matched by the wildcard. It matches to the characters **bicycle** alone, if a row exists with that description.

Query 2-33 narrows the search by adding another comparison condition that excludes a **manu_code** of **PRC**:

```
SELECT * FROM stock
  WHERE description LIKE 'bicycle%'
     AND manu_code NOT LIKE 'PRC'
     ORDER BY description, manu_code
```

Query 2-33

The statement retrieves only the rows found in Query Result 2-33:

stock_num	manu_code	description	unit_price	unit	unit_descr
102	SHM	bicycle brakes	\$220.00	case 4	sets/case
101	SHM	bicycle tires	\$68.00	box	4/box
105	SHM	bicycle wheels	\$80.00	pair	pair

Query Result 2-33

When you select from a large table and use an initial wildcard in the comparison string (such as '%cycle'), the query often takes longer to execute. This is because indexes cannot be used, so every row is searched.

MATCHES and Native Language Support

As indicated on page 2-25, you can set the DBNLS, LANG, and LC_COLLATE environment variables to influence NLS functionality regarding the collation order of data returned by a query.

Query 2-34 uses an ORDER BY clause and a WHERE clause specifying a matching range on a column containing data of type NCHAR. This NLS table, **abonnés**, is provided separately from the demonstration database. Query 2-34 is provided as the **sel_nls2.sql** command file with NLS-enabled **DB-Access**.

```
SELECT numéro,nom,prénom
  FROM abonnés
 WHERE nom MATCHES '[E-P]*'
 ORDER BY nom;
```

Query 2-34

The output in Query Result 2-34a and Query Result 2-34b show different rows selected for the MATCHES range by the same query, based on different NLS environment variable settings:

numéro	nom	prénom
13607	Hammer	Gerhard
13602	Hämmerle	Greta
13604	LaForêt	Jean-Noël
13610	LeMaître	Héloïse
13613	Llanero	Gloria Dolores
13603	Montaña	José Antonio
13611	Oatfield	Emily

Query Result 2-34a

In Query Result 2-34a, the rows for Étaix, Ötker, and Øverst are not selected and listed because, with standard ASCII ordering, the accented first letter of each name is not in the E-P MATCHES range for the **nom** column.

numéro	nom	prénom
13608	Étaix	Émile
13607	Hammer	Gerhard
13602	Hämmerle	Greta
13604	LaForêt	Jean-Noël
13610	LeMaître	Héloïse
13613	Llanero	Gloria Dolores
13603	Montaña	José Antonio
13611	Oatfield	Emily
13605	Ötker	Hans-Jürgen
13614	Øverst	Per-Anders

Query Result 2-34b

In Query Result 2-34b, the rows for Étaix, Ötker, and Øverst are included in the list because, with the appropriate NLS environment variable settings, the accented first letter of each name is in the E-P MATCHES range for the **nom** column.

Comparing for Special Characters

Query 2-35 uses the keyword `ESCAPE` with `LIKE` or `MATCHES` so you can protect a special character from misinterpretation as a wildcard symbol:

```
SELECT * FROM cust_calls
      WHERE res_descr LIKE '%!%%' ESCAPE '!'
```

Query 2-35

The `ESCAPE` keyword designates an *escape character* (it is an exclamation point [!] in this example), which protects the next character so it is interpreted as data and not as a wildcard. In the example, the escape character causes the middle percent sign to be treated as data. By using the `ESCAPE` keyword, you

can search for occurrences of a percent sign (%) in the **res_descr** column with the LIKE wildcard %. The query retrieves the row shown in Query Result 2-35:

```
customer_num    116
call_dtime      1992-12-21 11:24
user_id         mannyn
call_code       I
call_descr      Second complaint from this customer! Received
                 two cases right-handed outfielder gloves
                 (1 HRO) instead of one case lefties.
res_dtime       1992-12-27 08:19
res_descr       Memo to shipping (Ava Brown) to send case of
                 left-handed gloves, pick up wrong case; memo
                 to billing requesting 5% discount to placate
                 customer due to second offense and lateness
                 of resolution because of holiday
```

Query Result 2-35

Using Subscripting in a WHERE Clause

You can use *subscripting* in the WHERE clause of a SELECT statement to specify a range of characters or numbers in a column, as shown in Query 2-36:

```
SELECT catalog_num, stock_num, manu_code, cat_advert,
       cat_descr
FROM catalog
WHERE cat_advert[1,4] = 'High'
```

Query 2-36

The subscript [1,4] causes Query 2-36 to retrieve all rows in which the first four letters of the **cat_advert** column are High, as shown in Query Result 2-36:

```

catalog_num 10004
stock_num   2
manu_code    HRO
cat_advert   Highest Quality Ball Available, from
              Hand-Stitching to the Robinson Signature
cat_descr    Jackie Robinson signature ball. Highest professional quality, used by
              National League.

catalog_num 10005
stock_num   3
manu_code    HSK
cat_advert   High-Technology Design Expands the Sweet Spot
cat_descr    Pro-style wood. Available in sizes: 31, 32, 33, 34, 35.

catalog_num 10008
stock_num   4
manu_code    HRO
cat_advert   Highest Quality Football for High School and
              Collegiate Competitions
cat_descr    NFL-style, pigskin.

catalog_num 10012
stock_num   6
manu_code    SMT
cat_advert   High-Visibility Tennis, Day or Night
cat_descr    Soft yellow color for easy visibility in sunlight or
              artificial light.

catalog_num 10043
stock_num   202
manu_code    KAR
cat_advert   High-Quality Woods Appropriate for High School
              Competitions or Serious Amateurs
cat_descr    Full set of woods designed for precision control and
              power performance.

catalog_num 10045
stock_num   204
manu_code    KAR
cat_advert   High-Quality Beginning Set of Irons
              Appropriate for High School Competitions
cat_descr    Ideally balanced for optimum control. Nylon covered shaft.

catalog_num 10068
stock_num   310
manu_code    ANZ
cat_advert   High-Quality Kickboard
cat_descr    White. Standard size.

```

Query Result 2-36

Expressions and Derived Values

You are not limited to selecting columns by name. You can use the SELECT clause of a SELECT statement to perform computations on column data and to display information *derived* from the contents of one or more columns. You do this by listing an *expression* in the select list.

An expression consists of a column name, a constant, a quoted string, a keyword, or any combination of these items connected by operators. It also can include host variables (program data) when the SELECT statement is embedded in a program.

For more information, see the section “Expression,” in Chapter 1 of the *Informix Guide to SQL: Syntax*.

Arithmetic Expressions

An arithmetic expression contains at least one of the *arithmetic operators* listed in Figure 2-4 and produces a number. You cannot use TEXT or BYTE columns in arithmetic expressions.

Operator	Operation
+	addition
-	subtraction
*	multiplication
/	division

Figure 2-4 *Arithmetic operators used in arithmetic expressions*

Operations of this nature enable you to see the results of proposed computations without actually altering the data in the database. You can add an INTO TEMP clause to save the altered data in a temporary table for further reference, computations, or impromptu reports. Query 2-37 calculates a 7 percent sales tax on the **unit_price** column when the **unit_price** is \$400 or more (but does not update it in the database).

```
SELECT stock_num, description, unit, unit_descr,  
       unit_price, unit_price * 1.07  
FROM stock  
WHERE unit_price >= 400
```

Query 2-37

If you are using the **DB-Access** or **INFORMIX-SQL** Interactive Schema Editor, the result is displayed in a column labeled *expression*, as shown in Query Result 2-37:

stock_num	description	unit	unit_descr	unit_price	(expression)
1	baseball gloves	case 10	gloves/case	\$800.00	\$856.0000
1	baseball gloves	case 10	gloves/case	\$450.00	\$481.5000
4	football	case 24	/case	\$960.00	\$1027.2000
4	football	case 24	/case	\$480.00	\$513.6000
7	basketball	case 24	/case	\$600.00	\$642.0000
8	volleyball	case 24	/case	\$840.00	\$898.8000
102	bicycle brakes	case 4	sets/case	\$480.00	\$513.6000
111	10-spd, assmbld	each	each	\$499.99	\$534.9893
112	12-spd, assmbld	each	each	\$549.00	\$587.4300
113	18-spd, assmbld	each	each	\$685.90	\$733.9130
203	irons/wedge	case 2	sets/case	\$670.00	\$716.9000

Query Result 2-37

Query 2-38 calculates a surcharge of \$6.50 on orders when the quantity ordered is less than 5:

```
SELECT item_num, order_num, quantity,
       total_price, total_price + 6.50
FROM items
WHERE quantity < 5
```

Query 2-38

If you are using **DB-Access** or **INFORMIX-SQL**, the result is displayed in a column labeled *expression*, as shown in Query Result 2-38:

item_num	order_num	quantity	total_price	(expression)
1	1001	1	\$250.00	\$256.50
1	1002	1	\$960.00	\$966.50
2	1002	1	\$240.00	\$246.50
1	1003	1	\$20.00	\$26.50
2	1003	1	\$840.00	\$846.50
1	1004	1	\$250.00	\$256.50
2	1004	1	\$126.00	\$132.50
3	1004	1	\$240.00	\$246.50
4	1004	1	\$800.00	\$806.50
.				
.				
.				
1	1021	2	\$75.00	\$81.50
2	1021	3	\$225.00	\$231.50
3	1021	3	\$690.00	\$696.50
4	1021	2	\$624.00	\$630.50
1	1022	1	\$40.00	\$46.50
2	1022	2	\$96.00	\$102.50
3	1022	2	\$96.00	\$102.50
1	1023	2	\$40.00	\$46.50
2	1023	2	\$116.00	\$122.50
3	1023	1	\$80.00	\$86.50
4	1023	1	\$228.00	\$234.50
5	1023	1	\$170.00	\$176.50
6	1023	1	\$190.00	\$196.50

Query Result 2-38

Query 2-39 calculates and displays in an expression column (if you are using **DB-Access** or **INFORMIX-SQL**) the interval between when the customer call was received (`call_dtime`) and when the call was resolved (`res_dtime`), in days, hours, and minutes.

```
SELECT customer_num, user_id, call_code,
       call_dtime, res_dtime - call_dtime
FROM cust_calls
ORDER BY user_id
```

Query 2-39

customer_num	user_id	call_code	call_dtime	(expression)
116	manny	I	1992-12-21 11:24	5 20:55
116	manny	I	1992-11-28 13:34	0 03:13
106	maryj	D	1993-06-12 08:20	0 00:05
121	maryj	O	1993-07-10 14:05	0 00:01
127	maryj	I	1993-07-31 14:30	
110	richc	L	1993-07-07 10:24	0 00:06
119	richc	B	1993-07-01 15:00	0 17:21

Query Result 2-39

Using Display Labels

You can assign a *display label* to a computed or derived data column to replace the default column header *expression*. In Query Result 2-37, Query Result 2-38, and Query Result 2-39, the derived data is shown in a column called (expression). Query 2-40 also presents derived values, but the column displaying the derived values now has the descriptive header, `taxed`.

```
SELECT stock_num, description, unit, unit_descr,
       unit_price, unit_price * 1.07 taxed
FROM stock
WHERE unit_price >= 400
```

Query 2-40

Query Result 2-40 shows the label **taxed** is assigned to the expression in the select list that displays the results of the operation `unit_price * 1.07`:

stock_num	description	unit	unit_descr	unit_price	taxed
1	baseball gloves	case	10 gloves/case	\$800.00	\$856.0000
1	baseball gloves	case	10 gloves/case	\$450.00	\$481.5000
4	football	case	24/case	\$960.00	\$1027.2000
4	football	case	24/case	\$480.00	\$513.6000
7	basketball	case	24/case	\$600.00	\$642.0000
8	volleyball	case	24/case	\$840.00	\$898.8000
102	bicycle brakes	case	4 sets/case	\$480.00	\$513.6000
111	10-spd, assmbld	each	each	\$499.99	\$534.9893
112	12-spd, assmbld	each	each	\$549.00	\$587.4300
113	18-spd, assmbld	each	each	\$685.90	\$733.9130
203	irons/wedge	case	2 sets/case	\$670.00	\$716.9000

Query Result 2-40

In Query 2-41, the label **surcharge** is defined for the column that displays the results of the operation `total_price + 6.50`:

```
SELECT item_num, order_num, quantity,
       total_price, total_price + 6.50 surcharge
FROM items
WHERE quantity < 5
```

Query 2-41

The **surcharge** column is labeled in the output, as shown in Query Result 2-41:

item_num	order_num	quantity	total_price	surcharge
.				
.				
.				
2	1013	1	\$36.00	\$42.50
3	1013	1	\$48.00	\$54.50
4	1013	2	\$40.00	\$46.50
1	1014	1	\$960.00	\$966.50
2	1014	1	\$480.00	\$486.50
1	1015	1	\$450.00	\$456.50
1	1016	2	\$136.00	\$142.50
2	1016	3	\$90.00	\$96.50
3	1016	1	\$308.00	\$314.50
4	1016	1	\$120.00	\$126.50
1	1017	4	\$150.00	\$156.50
2	1017	1	\$230.00	\$236.50
.				
.				
.				

Query Result 2-41

Query 2-42 assigns the label **span** to the column that displays the results of subtracting the DATETIME column **call_dtime** from the DATETIME column **res_dtime**.

```
SELECT customer_num, user_id, call_code,
       call_dtime, res_dtime - call_dtime span
FROM cust_calls
ORDER BY user_id
```

Query 2-42

The **span** column is labeled in the output, as shown in Query Result 2-42:

customer_num	user_id	call_code	call_dtime	span
116	mannyn	I	1992-12-21 11:24	5 20:55
116	mannyn	I	1992-11-28 13:34	0 03:13
106	maryj	D	1993-06-12 08:20	0 00:05
121	maryj	O	1993-07-10 14:05	0 00:01
127	maryj	I	1993-07-31 14:30	
110	richc	L	1993-07-07 10:24	0 00:06
119	richc	B	1993-07-01 15:00	0 17:21

Query Result 2-42

Sorting on Derived Columns

When you want to use ORDER BY as an expression, you can use either the display label assigned to the expression or an integer, as shown in Query 2-43:

```
SELECT customer_num, user_id, call_code,  
       call_dtime, res_dtime - call_dtime span  
FROM cust_calls  
ORDER BY span
```

Query 2-43

Query 2-43 retrieves the same data from the **cust_calls** table as Query 2-42. In Query 2-43, the ORDER BY clause causes the data to be displayed in ascending order of the derived values in the **span** column, as shown in Query Result 2-43:

customer_num	user_id	call_code	call_dtime	span
127	maryj	I	1993-07-31 14:30	
121	maryj	O	1993-07-10 14:05	0 00:01
106	maryj	D	1993-06-12 08:20	0 00:05
110	richc	L	1993-07-07 10:24	0 00:06
116	mannyn	I	1992-11-28 13:34	0 03:13
119	richc	B	1993-07-01 15:00	0 17:21
116	mannyn	I	1992-12-21 11:24	5 20:55

Query Result 2-43

Query 2-44 uses an integer to represent the result of the operation **res_dtime - call_dtime** and retrieves the same rows that appear in Query Result 2-43.

```
SELECT customer_num, user_id, call_code,  
       call_dtime, res_dtime - call_dtime span  
FROM cust_calls  
ORDER BY 5
```

Query 2-44

Using Functions in SELECT Statements

In addition to column names and operators, an expression also can include one or more functions.

Expressions supported include aggregate, function (which include arithmetic functions), constant, and column expressions. These expressions are described in Chapter 1 of the *Informix Guide to SQL: Syntax*.

Aggregate Functions


The *aggregate* functions are COUNT, AVG, MAX, MIN, and SUM. They take on values that depend on all the rows selected and return information about rows, not the rows themselves. You cannot use these functions with TEXT or BYTE columns.

Aggregates often are used to summarize information about groups of rows in a table. This use is discussed in Chapter 3, “Composing Advanced SELECT Statements.” When you apply an aggregate function to an entire table, the result contains a single row that summarizes all of the selected rows.

Query 2-45 counts and displays the total number of rows in the **stock** table:

```
SELECT COUNT(*)  
FROM stock
```

Query 2-45



```
(count(*))  
  
73
```

Query Result 2-45

Query 2-46 includes a WHERE clause to count specific rows in the **stock** table, in this case, only those rows that have a **manu_code** of SHM:

```
SELECT COUNT (*)
  FROM stock
 WHERE manu_code = 'SHM'
```

Query 2-46

(count(*))
16

Query Result 2-46

By including the keyword DISTINCT (or its synonym UNIQUE) and a column name in Query 2-47, you can tally the number of different manufacturer codes in the **stock** table:

```
SELECT COUNT (DISTINCT manu_code)
  FROM stock
```

Query 2-47

(count)
9

Query Result 2-47

Query 2-48 computes the average **unit_price** of all rows in the **stock** table:

```
SELECT AVG (unit_price)
  FROM stock
```

Query 2-48

(avg)

Query Result 2-48

Query 2-49 computes the average **unit_price** of just those rows in the **stock** table that have a **manu_code** of SHM:

```
SELECT AVG (unit_price)
  FROM stock
 WHERE manu_code = 'SHM'
```

Query 2-49



(avg)

Query Result 2-49

You can combine aggregate functions in Query 2-50:

```
SELECT MAX (ship_charge), MIN (ship_charge)
  FROM orders
```

Query 2-50

Query 2-50 finds and displays both the highest and lowest **ship_charge** in the **orders** table, as shown in Query Result 2-50:



(max) (min)

Query Result 2-50

You can apply functions to expressions, and you can supply display labels for their results, as shown in Query 2-51:

```
SELECT MAX (res_dtime - call_dtime) maximum,
       MIN (res_dtime - call_dtime) minimum,
       AVG (res_dtime - call_dtime) average
  FROM cust_calls
```

Query 2-51

Query 2-51 finds and displays the maximum, minimum, and average amount of time (in days, hours, and minutes) between the reception and resolution of a customer call and labels the derived values appropriately. These amounts of time are shown in Query Result 2-51:

maximum	minimum	average
5 20:55	0 00:01	1 02:56

Query Result 2-51

Query 2-52 calculates the total ship_weight of orders shipped on July 13, 1993:

```
SELECT SUM (ship_weight)
FROM orders
WHERE ship_date = '07/13/1993'
```

Query 2-52

(sum)

Query Result 2-52

Time Functions

You can use the *time* functions DAY, MDY, MONTH, WEEKDAY, YEAR, and DATE in either the SELECT clause or the WHERE clause of a query. These functions return a value that corresponds to the expressions or arguments that you use to call the function. You also can use the CURRENT function to return a value with the current date and time, or the EXTEND function to adjust the precision of a DATE or DATETIME value.

Using DAY and CURRENT

Query 2-53 returns the day of the month for the call_dtime and res_dtime columns in two *expression* columns, as shown in Query Result 2-53:

```
SELECT customer_num, DAY (call_dtime), DAY (res_dtime)
       FROM cust_calls
```

Query 2-53

customer_num	(expression)	(expression)
106	12	12
110	7	7
119	1	2
121	10	10
127	31	
116	28	28
116	21	27

Query Result 2-53

Query 2-54 uses the DAY and CURRENT functions to compare column values to the current day of the month. It selects only those rows where the value is earlier than the current day.

```
SELECT customer_num, DAY (call_dtime), DAY (res_dtime)
       FROM cust_calls
      WHERE DAY (call_dtime) < DAY (CURRENT)
```

Query 2-54

customer_num	(expression)	(expression)
106	12	12
110	7	7
119	1	2
121	10	10

Query Result 2-54

Query 2-55 shows another use of the CURRENT function, selecting rows where the day is earlier than the current one:

```
SELECT customer_num, call_code, call_descr
  FROM cust_calls
 WHERE call_dtime < CURRENT YEAR TO DAY
```

Query 2-55

```
customer_num 106
call_code     D
call_descr    Order was received, but two of the cans of ANZ tennis
              balls within the case were empty

customer_num 116
call_code     I
call_descr    Received plain white swim caps (313 ANZ) instead of
              navy with team logo (313 SHM)

customer_num 116
call_code     I
call_descr    Second complaint from this customer! Received
              two cases right-handed outfielder gloves (1 HRO) instead of
              one case lefties.
```

Query Result 2-55

Using MONTH

Query 2-56 uses the MONTH function to extract and show what month the customer call was received and resolved and uses display labels for the resulting columns. However, it does not make a distinction between years.

```
SELECT customer_num,  
       MONTH (call_dtime) call_month,  
       MONTH (res_dtime) res_month  
FROM cust_calls
```

Query 2-56

customer_num	call_month	res_month
106	6	6
110	7	7
119	7	7
121	7	7
127	7	
116	11	11
116	12	12

Query Result 2-56

Query 2-57 uses the MONTH function plus DAY and CURRENT to show what month the customer call was received and resolved if DAY is earlier than the current day:

```
SELECT customer_num,  
       MONTH (call_dtime) called,  
       MONTH (res_dtime) resolved  
FROM cust_calls  
WHERE DAY (res_dtime) < DAY (CURRENT)
```

Query 2-57

customer_num	called	resolved
106	6	6
110	7	7
119	7	7
121	7	7

Query Result 2-57

Using WEEKDAY

In Query 2-58, the WEEKDAY function is used to indicate which day of the week calls are received and resolved (0 represents Sunday, 1 is Monday, and so on), and the expression columns are labeled:

```
SELECT customer_num,  
       WEEKDAY (call_dtime) called,  
       WEEKDAY (res_dtime) resolved  
FROM cust_calls  
ORDER BY resolved
```

Query 2-58

customer_num	called	resolved
127	3	
110	0	0
119	1	2
121	3	3
116	3	3
106	3	3
116	5	4

Query Result 2-58

Query 2-60 uses the COUNT and WEEKDAY functions to count how many calls were received on a weekend. This kind of statement can give you an idea of customer call patterns or indicate whether overtime pay might be required.

```
SELECT COUNT(*)  
FROM cust_calls  
WHERE WEEKDAY (call_dtime) IN (0,6)
```

Query 2-59

(count(*))

Query Result 2-59

Query 2-60 retrieves rows where the `call_dtime` is earlier than the beginning of the current year.

```
SELECT customer_num, call_code,
       YEAR (call_dtime) call_year,
       YEAR (res_dtime) res_year
FROM cust_calls
WHERE YEAR (call_dtime) < YEAR (TODAY)
```

Query 2-60

customer_num	call_code	call_year	res_year
116	I	1992	1992
116	I	1992	1992

Query Result 2-60

Formatting DATETIME Values

In Query 2-61, the `EXTEND` function restricts the two `DATETIME` values by displaying only the specified subfields:

```
SELECT customer_num,
       EXTEND (call_dtime, month to minute) call_time,
       EXTEND (res_dtime, month to minute) res_time
FROM cust_calls
ORDER BY res_time
```

Query 2-61

Query Result 2-61 returns the month-to-minute range for the columns labeled `call_time` and `res_time` and gives an indication of the workload:

customer_num	call_time	res_time
127	07-31 14:30	
106	06-12 08:20	06-12 08:25
119	07-01 15:00	07-02 08:21
110	07-07 10:24	07-07 10:30
121	07-10 14:05	07-10 14:06
116	11-28 13:34	11-28 16:47
116	12-21 11:24	12-27 08:19

Query Result 2-61

Using the DATE Function

Query 2-62 retrieves DATETIME values only when **call_dtime** is later than the specified DATE:

```
SELECT customer_num, call_dtime, res_dtime
FROM cust_calls
WHERE call_dtime > DATE ('12/31/92')
```

Query 2-62

Query Result 2-62 returns the following rows:

customer_num	call_dtime	res_dtime
106	1993-06-12 08:20	1993-06-12 08:25
110	1993-07-07 10:24	1993-07-07 10:30
119	1993-07-01 15:00	1993-07-02 08:21
121	1993-07-10 14:05	1993-07-10 14:06
127	1993-07-31 14:30	

Query Result 2-62

Query 2-63 converts DATETIME values to DATE format and displays the values, with labels, only when **call_dtime** is greater than or equal to the specified date.

```
SELECT customer_num,
       DATE (call_dtime) called,
       DATE (res_dtime) resolved
FROM cust_calls
WHERE call_dtime >= DATE ('1/1/93')
```

Query 2-63

customer_num	called	resolved
106	06/12/1993	06/12/1993
110	07/07/1993	07/07/1993
119	07/01/1993	07/02/1993
121	07/10/1993	07/10/1993
127	07/31/1993	

Query Result 2-63

Other Functions and Keywords

You also can use the LENGTH, USER, CURRENT, and TODAY functions anywhere in an SQL expression that you would use a constant. In addition, with **INFORMIX-OnLine Dynamic Server**, you can include the DBSERVER-NAME keyword in a SELECT statement to display the name of the database server where the current database resides.

You can use these functions and keywords to select an expression that consists entirely of constant values or an expression that includes column data. In the first instance, the result is the same for all rows of output.

In addition, you can use the HEX function to return the hexadecimal encoding of an expression, the ROUND function to return the rounded value of an expression, and the TRUNC function to return the truncated value of an expression.

In Query 2-64, the LENGTH function calculates the number of bytes in the combined **fname** and **lname** columns for each row where the length of **company** is greater than 15:

```
SELECT customer_num,  
       LENGTH (fname) + LENGTH (lname) namelength  
FROM customer  
WHERE LENGTH (company) > 15
```

Query 2-64

customer_num	namelength
101	11
105	13
107	11
112	14
115	11
118	10
119	10
120	10
122	12
124	11
125	10
126	12
127	10
128	11

Query Result 2-64

Although it might not be useful when you work with the **DB-Access** or **INFORMIX-SQL** Interactive Editor, the **LENGTH** function can be important to determine the string length for programs and reports. **LENGTH** returns the clipped length of a **CHARACTER** or **VARCHAR** string and the full number of bytes in a **TEXT** or **BYTE** string.

The **USER** function can be handy when you want to define a restricted view of a table that contains only your rows. For information on creating views, see Chapter 11, “Granting and Limiting Access to Your Database,” in this manual and the **GRANT** and **CREATE VIEW** statements in Chapter 1 of the *Informix Guide to SQL: Syntax*.

Query 2-65a specifies the **USER** function and the **cust_calls** table:

```
SELECT USER from cust_calls
```

Query 2-65a

Query 2-65b returns the user name (login account name) of the user who executes the query. It is repeated once for each row in the table.

```
SELECT * FROM cust_calls
      WHERE user_id = USER
```

Query 2-65b

If the user name of the current user is **richc**, Query 2-65b retrieves only those rows in the **cust_calls** table that are owned by that user.

```
customer_num  110
call_dtime    1993-07-07 10:24
user_id       richc
call_code     L
call_descr    Order placed one month ago (6/7) not received.
res_dtime     1993-07-07 10:30
res_descr     Checked with shipping (Ed Smith). Order sent
              yesterday- we were waiting for goods from ANZ. Next
              time will call with delay if necessary.

customer_num  119
call_dtime    1993-07-01 15:00
user_id       richc
call_code     B
call_descr    Bill does not reflect credit from previous order
res_dtime     1993-07-02 08:21
res_descr     Spoke with Jane Akant in Finance. She found the
              error and is sending new bill to customer
```

Query Result 2-65

Query 2-66, if issued when today's system date is July 10, 1993, returns this one row:

```
SELECT * FROM orders
      WHERE order_date = TODAY
```

Query 2-66

order_num	1018
order_date	07/10/1993
customer_num	121
ship_instruct	SW corner of Biltmore Mall
backlog	n
po_num	S22942
ship_date	07/13/1993
ship_weight	70.50
ship_charge	\$20.00
paid_date	08/06/1993

Query Result 2-66

You can include the keyword **DBSERVERNAME** (or its synonym, **SITENAME**) in a **SELECT** statement on **INFORMIX-OnLine Dynamic Server** to find the name of the database server. You can query on the **DBSERVERNAME** for any table that has rows, including system catalog tables.

In Query 2-67, you assign the label **server** to the **DBSERVERNAME** expression and also select the **tabid** column from the **systables** system catalog table. This table describes database tables, and **tabid** is the serial interval table identifier.

```
SELECT DBSERVERNAME server, tabid FROM systables
      WHERE tabid <= 4
```

Query 2-67

server	tabid
montague	1
montague	2
montague	3
montague	4

Query Result 2-67

Without the WHERE clause to restrict the values in the **tabid**, the database server name would be repeated for each row of the **systables** table.

In Query 2-68, the HEX function returns the hexadecimal format of three specified columns in the **customer** table:

```
SELECT HEX (customer_num) hexnum, HEX (zipcode) hexzip,  
       HEX (rowid) hexrow  
FROM customer
```

Query 2-68

hexnum	hexzip	hexrow
0x00000065	0x00016F86	0x00000001
0x00000066	0x00016FA5	0x00000002
0x00000067	0x0001705F	0x00000003
0x00000068	0x00016F4A	0x00000004
0x00000069	0x00016F46	0x00000005
0x0000006A	0x00016F6F	0x00000006
0x0000006B	0x00017060	0x00000007
0x0000006C	0x00016F6F	0x00000008
0x0000006D	0x00016F86	0x00000009
0x0000006E	0x00016F6E	0x0000000A
0x0000006F	0x00016F85	0x0000000B
0x00000070	0x00016F46	0x0000000C
0x00000071	0x00016F49	0x0000000D
0x00000072	0x00016F6E	0x0000000E
0x00000073	0x00016F49	0x0000000F
0x00000074	0x00016F58	0x00000010
0x00000075	0x00016F6F	0x00000011
0x00000076	0x00017191	0x00000012
0x00000077	0x00001F42	0x00000013
0x00000078	0x00014C18	0x00000014
0x00000079	0x00004DBA	0x00000015
0x0000007A	0x0000215C	0x00000016
0x0000007B	0x00007E00	0x00000017
0x0000007C	0x00012116	0x00000018
0x0000007D	0x00000857	0x00000019
0x0000007E	0x0001395B	0x0000001A
0x0000007F	0x0000EBF6	0x0000001B
0x00000080	0x00014C10	0x0000001C

Query Result 2-68

Using Stored Procedures in SELECT Statements

We have seen examples of SELECT statement expressions consisting of column names, operators, and functions. Another type of expression is one that contains a stored procedure call.

Stored procedures are procedures that contain special Stored Procedure Language (SPL) statements as well as SQL statements. For more information on stored procedures, refer to Chapter 14, “Creating and Using Stored Procedures.”

Stored procedures provide a way to extend the range of functions available; you can perform a subquery on each row you select.

For example, suppose you want a listing of the customer number, the customer’s last name, and the number of orders the customer has made. One way to retrieve this information is shown in Query 2-69. The **customer** table has **customer_num** and **lname** columns but no record of the number of orders each customer has made. You could write a **get_orders** procedure which queries the **orders** table for each **customer_num** and returns the number of corresponding orders (labeled **n_orders**).

```
SELECT customer_num, lname, get_orders(customer_num) n_orders
      FROM customer
```

Query 2-69

The output from this stored-procedure query is shown in Query Result 2-69:

customer_num	lname	n_orders
101	Pauli	1
102	Sadler	0
103	Currie	0
104	Higgins	4
105	Vector	0
106	Watson	2
107	Ream	0
108	Quinn	0
109	Miller	0
110	Jaeger	2
111	Keyes	1
112	Lawson	1
113	Beatty	0
114	Albertson	0
115	Grant	1
116	Parmelee	1
117	Sipes	2
118	Baxter	0
119	Shorter	1
120	Jewell	1
121	Wallack	1
122	O'Brian	1
123	Hanlon	1
124	Putnum	1
125	Henry	0
126	Neelie	1
127	Satifer	1
128	Lessor	0

Query Result 2-69

You can use stored procedures to encapsulate operations that you frequently perform in your queries. For example, the condition in Query 2-70 contains a procedure, **conv_price**, that converts the unit price of a stock item to a different currency and adds any import tariffs:

```
SELECT stock_num, manu_code, description FROM stock
      WHERE conv_price(unit_price, ex_rate = 1.50, tarif = 50.00)
            < 1000
```

Query 2-70

Multiple-Table SELECT Statements

You can select data from two or more tables by naming these tables in the FROM clause. Add a WHERE clause to create a *join* condition between at least one related column in each table. This creates a temporary composite table in which each pair of rows that satisfies the join condition is linked to form a single row.

A *simple join* combines information from two or more tables based on the relationship between one column in each table. A *composite join* is a join between two or more tables based on the relationship between two or more columns in each table.

To create a join, you must specify a relationship, called a *join condition*, between at least one column from each table. Because the columns are being compared, they must have compatible data types. When you join large tables, performance improves when you index the columns in the join condition.

Data types are described in Chapter 3 of the *Informix Guide to SQL: Reference*, and indexing is discussed in Chapter 10, “Tuning Your Data Model,” and Chapter 13, “Optimizing Your Queries.”

Creating a Cartesian Product

When you perform a multiple-table query that does not explicitly state a join condition among the tables, you create a *Cartesian product*. A Cartesian product consists of every possible combination of rows from the tables. This is usually a very large, unwieldy result and the data is inaccurate.

Query 2-71 selects from two tables and produces a Cartesian product:

```
SELECT * FROM customer, state
```

Query 2-71

Although only 52 rows exist in the **state** table and 28 rows in the **customer** table, the effect of Query 2-71 is to multiply the rows of one table by the rows of the other and retrieve an impractical 1456 rows, as shown in Query Result 2-71:

```
customer_num  101
fname         Ludwig
lname         Pauli
company       All Sports Supplies
address1      213 Erstwild Court
address2
city          Sunnyvale
state         CA
zipcode       94086
phone         408-789-8075
code          AK
sname         Alaska

customer_num  101
fname         Ludwig
lname         Pauli
company       All Sports Supplies
address1      213 Erstwild Court
address2
city          Sunnyvale
state         CA
zipcode       94086
phone         408-789-8075
code          HI
sname         Hawaii

customer_num  101
fname         Ludwig
lname         Pauli
company       All Sports Supplies
address1      213 Erstwild Court
address2
city          Sunnyvale
state         CA
zipcode       94086
phone         408-789-8075
code          CA
sname         California
.
.
.
```

Query Result 2-71

Note that some of the data displayed in the concatenated rows is inaccurate. For example, although the **city** and **state** from the **customer** table indicate an address in California, the **code** and **sname** from the **state** table might be for a different state.

Creating a Join

Conceptually, the first stage of any join is the creation of a Cartesian product. To refine or constrain this Cartesian product and eliminate meaningless rows of data, include a WHERE clause with a valid join condition in your SELECT statement.

This section illustrates *equi-joins*, *natural joins*, and *multiple-table joins*. Additional complex forms, such as *self-joins* and *outer joins*, are covered in Chapter 3, “Composing Advanced SELECT Statements.”

Equi-Join

An equi-join is a join based on equality or matching values. This equality is indicated with an equal sign (=) in the comparison operation in the WHERE clause, as shown in Query 2-72:

```
SELECT * FROM manufact, stock
        WHERE manufact.manu_code = stock.manu_code
```

Query 2-72

Query 2-72 joins the **manufact** and **stock** tables on the **manu_code** column, retrieving only those rows for which the values for the two columns are equal, as shown in Query Result 2-72:

```
manu_code    SMT
manu_name    Smith
lead_time    3
stock_num    1
manu_code    SMT
description   baseball gloves
unit_price    $450.00
unit         case
unit_descr   10 gloves/case

manu_code    SMT
manu_name    Smith
lead_time    3
stock_num    5
manu_code    SMT
description   tennis racquet
unit_price    $25.00
unit         each
unit_descr   each

manu_code    SMT
manu_name    Smith
lead_time    3
stock_num    6
manu_code    SMT
description   tennis ball
unit_price    $36.00
unit         case
unit_descr   24 cans/case

manu_code    ANZ
manu_name    Anza
lead_time    5
stock_num    5
manu_code    ANZ
description   tennis racquet
unit_price    $19.80
unit         each
unit_descr   each
.
.
.
```

Query Result 2-72

Note that in this equi-join, Query Result 2-72 includes the **manu_code** column from both the **manufact** and **stock** tables because the select list requested every column.

You also can create an equi-join with additional constraints, one where the comparison condition is based on the inequality of values in the joined columns. These joins use a relational operator other than = in the comparison condition specified in the WHERE clause.

When columns in the joined tables have the same name, the columns must be preceded by the name of a specific table and a period, as shown in Query 2-73:

```
SELECT order_num, order_date, ship_date, cust_calls.*
   FROM orders, cust_calls
  WHERE call_dtime >= ship_date
        AND cust_calls.customer_num = orders.customer_num
 ORDER BY customer_num
```

Query 2-73

Query 2-73 joins on the **customer_num** column and then selects only those rows where the **call_dtime** in the **cust_calls** table is greater than or equal to the **ship_date** in the **orders** table. It returns the rows shown in Query Result 2-73:

```

order_num      1004
order_date     05/22/1993
ship_date      05/30/1993
customer_num    106
call_dtime     1993-06-12 08:20
user_id        maryj
call_code      D
call_descr     Order received okay, but two of the cans of
               ANZ tennis balls within the case were empty
res_dtime      1993-06-12 08:25
res_descr      Authorized credit for two cans to customer,
               issued apology. Called ANZ buyer to report
               the qa problem.

order_num      1008
order_date     06/07/1993
ship_date      07/06/1993
customer_num    110
call_dtime     1993-07-07 10:24
user_id        richc
call_code      L
call_descr     Order placed one month ago (6/7) not received.
res_dtime      1993-07-07 10:30
res_descr      Checked with shipping (Ed Smith). Order out
               yesterday-was waiting for goods from ANZ.
               Next time will call with delay if necessary.

order_num      1023
order_date     07/24/1993
ship_date      07/30/1993
customer_num    127
call_dtime     1993-07-31 14:30
user_id        maryj
call_code      I
call_descr     Received Hero watches (item # 304) instead
               of ANZ watches
res_dtime
res_descr      Sent memo to shipping to send ANZ item 304
               to customer and pickup HRO watches. Should
               be done tomorrow, 8/1

```

Query Result 2-73

Natural Join

A natural join is structured so that the join column does not display data redundantly, as shown in Query 2-74:

```
SELECT manu_name, lead_time, stock.*  
      FROM manufact, stock  
      WHERE manufact.manu_code = stock.manu_code
```

Query 2-74

Like the example for equi-join, Query 2-74 joins the **manufact** and **stock** tables on the **manu_code** column. Because the select list is more closely defined, the **manu_code** is listed only once for each row retrieved, as shown in Query Result 2-74:

```
manu_name    Smith
lead_time    3
stock_num    1
manu_code    SMT
description   baseball gloves
unit_price   $450.00
unit         case
unit_descr   10 gloves/case

manu_name    Smith
lead_time    3
stock_num    5
manu_code    SMT
description   tennis racquet
unit_price   $25.00
unit         each
unit_descr   each

manu_name    Smith
lead_time    3
stock_num    6
manu_code    SMT
description   tennis ball
unit_price   $36.00
unit         case
unit_descr   24 cans/case

manu_name    Anza
lead_time    5
stock_num    5
manu_code    ANZ
description   tennis racquet
unit_price   $19.80
unit         each
unit_descr   each
.
.
.
```

Query Result 2-74

All joins are *associative*, that is, the order of the joining terms in the WHERE clause does not affect the meaning of the join.

Both of the statements in Query 2-75 create the same natural join:

```
SELECT catalog.*, description, unit_price, unit, unit_descr
FROM catalog, stock
WHERE catalog.stock_num = stock.stock_num
      AND catalog.manu_code = stock.manu_code
      AND catalog_num = 10017

SELECT catalog.*, description, unit_price, unit, unit_descr
FROM catalog, stock
WHERE catalog_num = 10017
      AND catalog.manu_code = stock.manu_code
      AND catalog.stock_num = stock.stock_num
```

Query 2-75

Each statement retrieves the row shown in Query Result 2-75:

```
catalog_num  10017
stock_num    101
manu_code    PRC
cat_descr    Reinforced, hand-finished tubular. Polyurethane belted.
              Effective against punctures. Mixed tread for super wear
              and road grip.
cat_picture   <BYTE value>

cat_advert    Ultimate in Puncture Protection, Tires
              Designed for In-City Riding
description   bicycle tires
unit_price    $88.00
unit          box
unit_descr    4/box
```

Query Result 2-75

Note that Query Result 2-75 includes a TEXT column, **cat_descr**; a BYTE column, **cat_picture**; and a VARCHAR column, **cat_advert**.

Multiple-Table Join

A multiple-table join connects more than two tables on one or more associated columns; it can be an equi-join or a natural join.

Query 2-76 creates an equi-join on the **catalog**, **stock**, and **manufact** tables and retrieves the following row:

```
SELECT * FROM catalog, stock, manufact
      WHERE catalog.stock_num = stock.stock_num
            AND stock.manu_code = manufact.manu_code
            AND catalog_num = 10025
```

Query 2-76

Query Result 2-76 retrieves the rows shown in Query 2-76:

```
catalog_num  10025
stock_num    106
manu_code    PRC
cat_descr    Hard anodized alloy with pearl finish; 6mm hex bolt hardware.
              Available in lengths of 90-140mm in 10mm increments.
cat_picture  <BYTE value>

cat_advert   ProCycle Stem with Pearl Finish
stock_num    106
manu_code    PRC
description   bicycle stem
unit_price   $23.00
unit         each
unit_descr   each
manu_code    PRC
manu_name    ProCycle
lead_time    9
```

Query Result 2-76

Note that the **manu_code** is repeated three times, once for each table, and **stock_num** is repeated twice.

Because of the considerable duplication of a multiple-table query in Query 2-76, it is wise to more closely define the SELECT statement by including specific columns in the select list, as shown in Query 2-77:

```
SELECT catalog.*, description, unit_price, unit,
      unit_descr, manu_name, lead_time
FROM catalog, stock, manufact
WHERE catalog.stock_num = stock.stock_num
      AND stock.manu_code = manufact.manu_code
      AND catalog_num = 10025
```

Query 2-77

Query 2-77 uses a wildcard to select all columns from the table having the most columns and then specifies columns from the other two tables. It produces the natural join, shown in Query Result 2-77, that displays the same information as the previous example, but without duplication:

```
catalog_num  10025
stock_num    106
manu_code    PRC
cat_descr    Hard anodized alloy with pearl finish. 6mm hex bolt hardware.
              Available in lengths of 90-140mm in 10mm increments.
cat_picture  <BYTE value>

cat_advert   ProCycle Stem with Pearl Finish
description  bicycle stem
unit_price   $23.00
unit         each
unit_descr   each
manu_name     ProCycle
lead_time     9
```

Query Result 2-77

Some Query Shortcuts

You can use aliases, the INTO TEMP clause, and display labels to speed your way through joins and multiple-table queries and to produce output for other uses.

Using Aliases

You can make multiple-table queries shorter and more readable by assigning *aliases* to the tables in a SELECT statement. An alias is a word that immediately follows the name of a table in the FROM clause. You can use it wherever the table name would be used, for instance, as a prefix to the column names in the other clauses.

```
SELECT s.stock_num, s.manu_code, s.description,
       s.unit_price, s.unit, c.catalog_num,
       c.cat_descr, c.cat_advert, m.lead_time
FROM stock s, catalog c, manufact m
WHERE s.stock_num = c.stock_num
      AND s.manu_code = c.manu_code
      AND s.manu_code = m.manu_code
      AND s.manu_code IN ('HRO', 'HSK')
      AND s.stock_num BETWEEN 100 AND 301
ORDER BY catalog_num
```

Query 2-78a

The associative nature of the SELECT statement allows you to use an alias before you define it. In Query 2-78a, the aliases **s** for the **stock** table, **c** for the **catalog** table, and **m** for the **manufact** table are specified in the FROM clause and used throughout the SELECT and WHERE clauses as column prefixes.

Compare the length of Query 2-78a with Query 2-78b, which does not use aliases:

```
SELECT stock.stock_num, stock.manu_code, stock.description,  
       stock.unit_price, stock.unit, catalog.catalog_num,  
       catalog.cat_descr, catalog.cat_advert,  
       manufact.lead_time  
FROM stock, catalog, manufact  
WHERE stock.stock_num = catalog.stock_num  
      AND stock.manu_code = catalog.manu_code  
      AND stock.manu_code = manufact.manu_code  
      AND stock.manu_code IN ('HRO', 'HSK')  
      AND stock.stock_num BETWEEN 100 AND 301  
ORDER BY catalog_num
```

Query 2-78b

Query 2-78a and Query 2-78b are equivalent and retrieve the data shown in Query Result 2-78:

```

stock_num      110
manu_code      HRO
description     helmet
unit_price     $260.00
unit           case
catalog_num    10033
cat_descr      Newest ultralight helmet uses plastic shell. Largest ventilation
channels of any helmet on the market. 8.5 oz.
cat_advert     Lightweight Plastic Slatted with Vents Assures Cool
Comfort Without Sacrificing Protection
lead_time      4

stock_num      110
manu_code      HSK
description     helmet
unit_price     $308.00
unit           each
catalog_num    10034
cat_descr      Aerodynamic (teardrop) helmet covered with anti-drag fabric.
Credited with shaving 2 seconds/mile from winner's time in
Tour de France time-trial. 7.5 oz.
cat_advert     Teardrop Design Endorsed by Yellow Jerseys,
You Can Time the Difference
lead_time      5

stock_num      205
manu_code      HRO
description     3 golf balls
unit_price     $312.00
unit           each
catalog_num    10048
cat_descr      Combination fluorescent yellow and standard white.
cat_advert     HiFlier Golf Balls: Case Includes Fluorescent
Yellow and Standard White
lead_time      4

stock_num      301
manu_code      HRO
description     running shoes
unit_price     $42.50
unit           each
catalog_num    10050
cat_descr      Engineered for serious training with exceptional stability.
Fabulous shock absorption. Great durability. Specify
mens/womens, size.
cat_advert     Pronators and Supinators Take Heart: A Serious
Training Shoe For Runners Who Need Motion Control
lead_time      4

```

Query Result 2-78

Note that you cannot ORDER BY the TEXT column **cat_descr** or the BYTE column **cat_picture**.

You also can use aliases to shorten your queries on external tables residing in external databases.

Query 2-79 joins columns from two tables that reside in different databases and systems, neither of which is the current database or system:

```
SELECT order_num, lname, fname, phone
      FROM masterdb@central:customer c, sales@western:orders o
      WHERE c.customer_num = o.customer_num
            AND order_num <= 1010
```

Query 2-79

By assigning the aliases **c** and **o** to the long *database@system:table* names, **masterdb@central:customer** and **sales@western:orders**, respectively, you can use the aliases to shorten the expression in the WHERE clause and retrieve the data shown in Query Result 2-79:

order_num	lname	fname	phone
1001	Higgins	Anthony	415-368-1100
1002	Pauli	Ludwig	408-789-8075
1003	Higgins	Anthony	415-368-1100
1004	Watson	George	415-389-8789
1005	Parmelee	Jean	415-534-8822
1006	Lawson	Margaret	415-887-7235
1007	Sipes	Arnold	415-245-4578
1008	Jaeger	Roy	415-743-3611
1009	Keyes	Frances	408-277-7245
1010	Grant	Alfred	415-356-1123

Query Result 2-79

For more information on external tables and external databases, see Chapter 12, “Understanding Informix Networking,” in this manual and Chapter 1 in the *Informix Guide to SQL: Syntax*.

You also can use *synonyms* as shorthand references to the long names of external and current tables and views. For details on how to create and use synonyms, see Chapter 12, “Understanding Informix Networking,” in this manual and the CREATE SYNONYM statement in Chapter 1 of the *Informix Guide to SQL: Syntax*.

The INTO TEMP Clause

By adding an INTO TEMP clause to your SELECT statement, you can temporarily save the results of a multiple-table query in a separate table that can be queried or manipulated without modifying the database. Temporary tables are dropped when you end your SQL session or when your program or report terminates.

Query 2-80 creates a temporary table called **stockman** and stores the results of the query in it. Because all columns in a temp table must have names, the alias **adj_price** is required.

```
SELECT DISTINCT stock_num, manu_name, description,
               unit_price, unit_price * 1.05 adj_price
FROM stock, manufact
WHERE manufact.manu_code = stock.manu_code
INTO TEMP stockman
```

Query 2-80

stock_num	manu_name	description	unit_price	adj_price
1	Hero	baseball gloves	\$250.00	\$262.5000
1	Husky	baseball gloves	\$800.00	\$840.0000
1	Smith	baseball gloves	\$450.00	\$472.5000
2	Hero	baseball	\$126.00	\$132.3000
3	Husky	baseball bat	\$240.00	\$252.0000
4	Hero	football	\$480.00	\$504.0000
4	Husky	football	\$960.00	\$1008.0000
.
306	Shimara	tandem adapter	\$190.00	\$199.5000
307	ProCycle	infant jogger	\$250.00	\$262.5000
308	ProCycle	twin jogger	\$280.00	\$294.0000
309	Hero	ear drops	\$40.00	\$42.0000
309	Shimara	ear drops	\$40.00	\$42.0000
310	Anza	kick board	\$84.00	\$88.2000
310	Shimara	kick board	\$80.00	\$84.0000
311	Shimara	water gloves	\$48.00	\$50.4000
312	Hero	racer goggles	\$72.00	\$75.6000
312	Shimara	racer goggles	\$96.00	\$100.8000
313	Anza	swim cap	\$60.00	\$63.0000
313	Shimara	swim cap	\$72.00	\$75.6000

Query Result 2-80

You can query on this table and join it with other tables, which avoids a multiple sort and lets you move more quickly through the database. Temporary tables are discussed at greater length in Chapter 13, “Optimizing Your Queries.”

Summary

This chapter introduced sample syntax and results for basic kinds of `SELECT` statements that are used to query on a relational database. Earlier sections of the chapter showed how to perform the following activities:

- Select all columns and rows from a table with the `SELECT` and `FROM` clauses.
- Select specific columns from a table with the `SELECT` and `FROM` clauses.
- Select specific rows from a table with the `SELECT`, `FROM`, and `WHERE` clauses.
- Use the `DISTINCT` or `UNIQUE` keyword in the `SELECT` clause to eliminate duplicate rows from query results.
- Sort retrieved data with the `ORDER BY` clause and the `DESC` keyword.
- Select and order data containing foreign characters.
- Use the `BETWEEN`, `IN`, `MATCHES`, and `LIKE` keywords and various relational operators in the `WHERE` clause to create a comparison condition.
- Create comparison conditions that include values, exclude values, find a range of values (with keywords, relational operators, and subscripting), and find a subset of values.
- Perform variable text searches using exact text comparisons, variable-length wildcards, and restricted and unrestricted wildcards.
- Use the logical operators `AND`, `OR`, and `NOT` to connect search conditions or Boolean expressions in a `WHERE` clause.
- Use the `ESCAPE` keyword to protect special characters in a query.
- Search for null values with the `IS NULL` and `IS NOT NULL` keywords in the `WHERE` clause.
- Use arithmetic operators in the `SELECT` clause to perform computations on number fields and display derived data.
- Use substrings and subscripting to tailor your queries.
- Assign display labels to computed columns as a formatting tool for reports.
- Use the aggregate functions `COUNT`, `AVG`, `MAX`, `MIN`, and `SUM` in the `SELECT` clause to calculate and retrieve specific data.
- Include the time functions `DATE`, `DAY`, `MDY`, `MONTH`, `WEEKDAY`, `YEAR`, `CURRENT`, and `EXTEND` plus the `TODAY`, `LENGTH`, and `USER` functions in your `SELECT` statements.
- Include stored procedures in your `SELECT` statements.

This chapter also introduced simple join conditions that enable you to select and display data from two or more tables. The section “Multiple-Table SELECT Statements” described how to perform the following actions:

- Create a Cartesian product.
- Constrain a Cartesian product by including a WHERE clause with a valid join condition in your query.
- Define and create a natural join and an equi-join.
- Join two or more tables on one or more columns.
- Use aliases as a shortcut in multiple-table queries.
- Retrieve selected data into a separate, temporary table with the INTO TEMP clause to perform computations outside the database.

The next chapter explains more complex queries and subqueries; self-joins and outer joins; the GROUP BY and HAVING clauses; and the UNION, INTERSECTION, and DIFFERENCE set operations.

Summary

Composing Advanced SELECT Statements

Chapter Overview	3
Using the GROUP BY and HAVING Clauses	4
Using the GROUP BY Clause	4
Using the HAVING Clause	9
Creating Advanced Joins	11
Self-Joins	11
Outer Joins	21
Simple Join	22
Simple Outer Join on Two Tables	24
Outer Join for a Simple Join to a Third Table	26
Outer Join for an Outer Join to a Third Table	28
Outer Join of Two Tables to a Third Table	30
Subqueries in SELECT Statements	32
Using ALL	33
Using ANY	34
Single-Valued Subqueries	36
Correlated Subqueries	37
Using EXISTS	38
Set Operations	42
Union	43
Intersection	51
Difference	53
Summary	54



Chapter Overview

Chapter 2, “Composing Simple SELECT Statements,” demonstrated some of the basic ways to retrieve data from a relational database with the SELECT statement. This chapter increases the scope of what you can do with this powerful SQL statement and enables you to perform more complex database queries and data manipulation.

Whereas the previous chapter focused on five of the clauses in SELECT statement syntax, this chapter adds two more. You can use the GROUP BY clause with aggregate functions to organize rows returned by the FROM clause. You can include a HAVING clause to place conditions on the values returned by the GROUP BY clause.

This chapter extends the earlier discussion of joins. It illustrates *self-joins*, which enable you to join a table to itself, and four kinds of *outer joins*, where you apply the keyword OUTER to treat two or more joined tables unequally. It also introduces correlated and uncorrelated subqueries and their operational keywords, shows how to combine queries with the UNION operator, and defines the set operations known as union, intersection, and difference.

Examples in this chapter show how to use some or all of the SELECT statement clauses in your queries. The clauses must appear in the following order:

1. SELECT
2. FROM
3. WHERE
4. GROUP BY
5. HAVING
6. ORDER BY
7. INTO TEMP

An additional SELECT statement clause, INTO, which you can use to specify program and host variables in **INFORMIX-4GL** and SQL APIs is described in Chapter 5, “Programming with SQL,” as well as in the manuals that come with the product.

Using the GROUP BY and HAVING Clauses

The optional GROUP BY and HAVING clauses add functionality to your SELECT statement. You can include one or both in a basic SELECT statement to increase your ability to manipulate aggregates.

The GROUP BY clause combines similar rows, producing a single result row for each *group* of rows that have the same values for each column listed in the select list. The HAVING clause sets conditions on those groups after you form them. You can use a GROUP BY clause without a HAVING clause, or a HAVING clause without a GROUP BY clause.

Using the GROUP BY Clause

The GROUP BY clause divides a table into sets. It is most often combined with aggregate functions that produce summary values for each of those sets. Some of the examples in Chapter 2, “Composing Simple SELECT Statements” show the use of aggregate functions applied to a whole table. This chapter illustrates aggregate functions applied to groups of rows.

Using the GROUP BY clause without aggregates is much like using the DISTINCT (or UNIQUE) keyword in the SELECT clause. Chapter 2, “Composing Simple SELECT Statements” included the statement found in Query 3-1a:

```
SELECT DISTINCT customer_num FROM orders
```

Query 3-1a

You also could write the statement as shown in Query 3-1b:

```
SELECT customer_num
FROM orders
GROUP BY customer_num
```

Query 3-1b

Query 3-1a and Query 3-1b return the rows shown in Query Result 3-1:

customer_num
101
104
106
110
111
112
115
116
117
119
120
121
122
123
124
126
127

Query Result 3-1

The GROUP BY clause collects the rows into sets so that each row in each set has equal customer numbers. With no other columns selected, the result is a list of the unique **customer_num** values.

The power of the GROUP BY clause is more apparent when you use it with aggregate functions.

Query 3-2 retrieves the number of items and the total price of all items for each order:

```
SELECT order_num, COUNT (*) number, SUM (total_price) price
FROM items
GROUP BY order_num
```

Query 3-2

The GROUP BY clause causes the rows of the **items** table to be collected into groups, each group composed of rows that have identical **order_num** values (that is, the items of each order are grouped together). After you form the groups, the aggregate functions COUNT and SUM are applied within each group.

Query 3-2 returns one row for each group. It uses labels to give names to the results of the COUNT and SUM expressions, as shown in Query Result 3-2:

order_num	number	price
1001	1	\$250.00
1002	2	\$1200.00
1003	3	\$959.00
1004	4	\$1416.00
1005	4	\$562.00
1006	5	\$448.00
1007	5	\$1696.00
1008	2	\$940.00
.		
.		
.		
1015	1	\$450.00
1016	4	\$654.00
1017	3	\$584.00
1018	5	\$1131.00
1019	1	\$1499.97
1020	2	\$438.00
1021	4	\$1614.00
1022	3	\$232.00
1023	6	\$824.00

Query Result 3-2

Query Result 3-2 collects the rows of the **items** table into groups that have identical order numbers and computes the COUNT of rows in each group and the sum of the prices.

Note that you cannot include a column having a TEXT or BYTE data type in a GROUP BY clause. To *group*, you must be able to *sort*, and no natural sort order exists for TEXT or BYTE data.

Unlike the ORDER BY clause, the GROUP BY clause does not order data. Include an ORDER BY clause *after* your GROUP BY clause if you want to sort data in a particular order or to sort on an aggregate in the select list.

Query 3-3 is the same as Query 3-2 but includes an ORDER BY clause to sort the retrieved rows in ascending order of **price**, as shown in Query Result 3-3:

```
SELECT order_num, COUNT(*) number, SUM (total_price) price
FROM items
GROUP BY order_num
ORDER BY price
```

Query 3-3

order_num	number	price
1010	2	\$84.00
1011	1	\$99.00
1013	4	\$143.80
1022	3	\$232.00
1001	1	\$250.00
1020	2	\$438.00
1006	5	\$448.00
1015	1	\$450.00
1009	1	\$450.00
.		
.		
.		
1018	5	\$1131.00
1002	2	\$1200.00
1004	4	\$1416.00
1014	2	\$1440.00
1019	1	\$1499.97
1021	4	\$1614.00
1007	5	\$1696.00

Query Result 3-3

As stated in Chapter 2, “Composing Simple SELECT Statements,” you can use an integer in an ORDER BY clause to indicate the position of a column in the select list. You also can use an integer in a GROUP BY clause to indicate the position of column names or display labels in the group list.

Query 3-3 returns the same rows as Query 3-4, as shown in Query Result 3-3.

```
SELECT order_num, COUNT(*) number, SUM (total_price) price
FROM items
GROUP BY 1
ORDER BY 3
```

Query 3-4

When you build a query, remember that all nonaggregate columns that are in the select list in the SELECT clause must also be included in the group list in the GROUP BY clause. The reason for this is that a SELECT with GROUP BY must return only one row per group. Columns that are listed after GROUP BY are certain to reflect only one distinct value within a group, and that value can be returned. However, a column not listed after GROUP BY might contain different values in the rows that are contained in a group.

As shown in Query 3-5, you can use the GROUP BY clause in a SELECT statement that joins tables:

```
SELECT o.order_num, SUM (i.total_price)
FROM orders o, items i
WHERE o.order_date > '01/01/92'
      AND o.customer_num = 110
      AND o.order_num = i.order_num
GROUP BY o.order_num
```

Query 3-5

Query 3-5 joins the **orders** and **items** tables, assigns table aliases to them, and returns the rows shown in Query Result 3-5:

order_num	(sum)
1008	\$940.00
1015	\$450.00

Query Result 3-5

Using the HAVING Clause

The HAVING clause usually complements a GROUP BY clause by applying one or more qualifying conditions to groups after they are formed, which is similar to the way the WHERE clause qualifies individual rows. One advantage to using a HAVING clause is that you can include aggregates in the search condition, whereas you cannot include aggregates in the search condition of a WHERE clause.

Each HAVING condition compares one column or aggregate expression of the group with another aggregate expression of the group or with a constant. You can use HAVING to place conditions on both column values and aggregate values in the group list.

Query 3-6 returns the average total price per item on all orders that have more than two items. The HAVING clause tests each group as it is formed and selects those composed of two or more rows.

```
SELECT order_num, COUNT(*) number, AVG (total_price) average
FROM items
GROUP BY order_num
HAVING COUNT(*) > 2
```

Query 3-6

order_num	number	average
1003	3	\$319.67
1004	4	\$354.00
1005	4	\$140.50
1006	5	\$89.60
1007	5	\$339.20
1013	4	\$35.95
1016	4	\$163.50
1017	3	\$194.67
1018	5	\$226.20
1021	4	\$403.50
1022	3	\$77.33
1023	6	\$137.33

Query Result 3-6

If you use a HAVING clause without a GROUP BY clause, the HAVING condition applies to all rows that satisfy the search condition. In other words, all rows that satisfy the search condition make up a single group.

Query 3-7, a modified version of Query 3-6, returns just one row, the average of all **total_price** values in the table:

```
SELECT AVG (total_price) average
FROM items
HAVING count(*) > 2
```

Query 3-7

average
\$270.97

Query Result 3-7

If Query 3-7, like Query 3-6, had included the nonaggregate column **order_num** in the select list, you would have to include a GROUP BY clause with that column in the group list. In addition, if the condition in the HAVING clause was not satisfied, the output would show the column heading and a message would indicate that no rows were found.

Query 3-8 contains all the SELECT statement clauses that you can use in the Informix version of interactive SQL (the INTO clause naming program or host variables is available only in an **INFORMIX-4GL** or an SQL API:

```
SELECT o.order_num, SUM (i.total_price) price,
       paid_date - order_date span
FROM orders o, items i
WHERE o.order_date > '01/01/92'
      AND o.customer_num > 110
      AND o.order_num = i.order_num
GROUP BY 1, 3
HAVING COUNT (*) < 5
ORDER BY 3
INTO TEMP temptabl
```

Query 3-8

Query 3-8 joins the **orders** and **items** tables; employs display labels, table aliases, and integers used as column indicators; groups and orders the data; and puts the following results in a temporary table, as shown in Query Result 3-8:

order_num	price	span
1017	\$584.00	
1016	\$654.00	
1012	\$1040.00	
1019	\$1499.97	26
1005	\$562.00	28
1021	\$1614.00	30
1022	\$232.00	40
1010	\$84.00	66
1009	\$450.00	68
1020	\$438.00	71

Query Result 3-8

Creating Advanced Joins

Chapter 2, “Composing Simple SELECT Statements,” showed how to include a WHERE clause in a SELECT statement to join two or more tables on one or more columns. It illustrated natural joins and equi-joins.

This chapter discusses the uses of two more complex kinds of joins: self-joins and outer joins. As described for simple joins, you can define aliases for tables and assign display labels to expressions to shorten your multiple-table queries. You can also sort data with an ORDER BY clause and SELECT query results into a temporary table.

Self-Joins

A join does not always have to involve two different tables. You can join a table to itself, creating a *self-join*. This can be useful when you want to compare values in a column to other values in the same column.

To create a self-join, list a table twice in the FROM clause, assigning it a different alias each time. Use the aliases to refer to the table in the SELECT and WHERE clauses as if it were two separate tables. (Aliases in SELECT statements are shown in Chapter 2, “Composing Simple SELECT Statements” of this manual and discussed in Chapter 1 of the *Informix Guide to SQL: Syntax*.)

Just as in joins between tables, you can use arithmetic expressions in self-joins. You can test for null values, and you can ORDER BY a specified column in ascending or descending order.

Query 3-9 finds pairs of orders where the **ship_weight** differs by a factor of five or more and the **ship_date** is not null, and orders the data by **ship_date**:

```
SELECT x.order_num, x.ship_weight, x.ship_date,
       y.order_num, y.ship_weight, y.ship_date
FROM orders x, orders y
WHERE x.ship_weight >= 5 * y.ship_weight
      AND x.ship_date IS NOT NULL
      AND y.ship_date IS NOT NULL
ORDER BY x.ship_date
```

Query 3-9

order_num	ship_weight	ship_date	order_num	ship_weight	ship_date
1004	95.80	05/30/1993	1011	10.40	07/03/1993
1004	95.80	05/30/1993	1020	14.00	07/16/1993
1004	95.80	05/30/1993	1022	15.00	07/30/1993
1007	125.90	06/05/1993	1015	20.60	07/16/1993
1007	125.90	06/05/1993	1020	14.00	07/16/1993
1007	125.90	06/05/1993	1022	15.00	07/30/1993
1007	125.90	06/05/1993	1011	10.40	07/03/1993
1007	125.90	06/05/1993	1001	20.40	06/01/1993
1007	125.90	06/05/1993	1009	20.40	06/21/1993
1005	80.80	06/09/1993	1011	10.40	07/03/1993
1005	80.80	06/09/1993	1020	14.00	07/16/1993
1005	80.80	06/09/1993	1022	15.00	07/30/1993
1012	70.80	06/29/1993	1011	10.40	07/03/1993
1012	70.80	06/29/1993	1020	14.00	07/16/1993
1013	60.80	07/10/1993	1011	10.40	07/03/1993
1017	60.00	07/13/1993	1011	10.40	07/03/1993
1018	70.50	07/13/1993	1011	10.40	07/03/1993
.					
.					
.					

Query Result 3-9

Suppose you want to select the results of a self-join into a temporary table. You would, of course, append an INTO TEMP clause to the SELECT statement. However, because you are, in effect, creating a new table, you also must rename at least one set of column names by assigning them display labels. Otherwise, you get an error message that indicates duplicate column names, and the temporary table is not created.

Query 3-10, which is similar to Query 3-9, labels all columns selected from the **orders** table and puts them in a temporary table called **shipping**:

```
SELECT x.order_num orders1, x.po_num purch1,
       x.ship_date ship1, y.order_num orders2,
       y.po_num purch2, y.ship_date ship2
FROM orders x, orders y
WHERE x.ship_weight >= 5 * y.ship_weight
      AND x.ship_date IS NOT NULL
      AND y.ship_date IS NOT NULL
ORDER BY orders1, orders2
INTO TEMP shipping
```

Query 3-10

If you **SELECT *** from that table, you see the rows in Query Result 3-10:

orders1	purch1	ship1	orders2	purch2	ship2
1004	8006	05/30/1993	1011	B77897	07/03/1993
1004	8006	05/30/1993	1020	W2286	07/16/1993
1004	8006	05/30/1993	1022	W9925	07/30/1993
1005	2865	06/09/1993	1011	B77897	07/03/1993
1005	2865	06/09/1993	1020	W2286	07/16/1993
1005	2865	06/09/1993	1022	W9925	07/30/1993
1007	278693	06/05/1993	1001	B77836	06/01/1993
1007	278693	06/05/1993	1009	4745	06/21/1993
1007	278693	06/05/1993	1011	B77897	07/03/1993
1007	278693	06/05/1993	1015	MA003	07/16/1993
1007	278693	06/05/1993	1020	W2286	07/16/1993
1007	278693	06/05/1993	1022	W9925	07/30/1993
1012	278701	06/29/1993	1011	B77897	07/03/1993
1012	278701	06/29/1993	1020	W2286	07/16/1993
1013	B77930	07/10/1993	1011	B77897	07/03/1993
1017	DM354331	07/13/1993	1011	B77897	07/03/1993
1018	S22942	07/13/1993	1011	B77897	07/03/1993
1018	S22942	07/13/1993	1020	W2286	07/16/1993
1019	Z55709	07/16/1993	1011	B77897	07/03/1993
1019	Z55709	07/16/1993	1020	W2286	07/16/1993
1019	Z55709	07/16/1993	1022	W9925	07/30/1993
1023	KF2961	07/30/1993	1011	B77897	07/03/1993

Query Result 3-10

You can join a table to itself more than once. The maximum number of self-joins depends on the resources available to you.

The self-join in Query 3-11 creates a list of those items in the **stock** table that are supplied by three manufacturers. By including the last two conditions in the WHERE clause, it eliminates duplicate manufacturer codes in rows retrieved.

```
SELECT s1.manu_code, s2.manu_code, s3.manu_code,
       s1.stock_num, s1.description
FROM stock s1, stock s2, stock s3
WHERE s1.stock_num = s2.stock_num
      AND s2.stock_num = s3.stock_num
      AND s1.manu_code < s2.manu_code
      AND s2.manu_code < s3.manu_code
ORDER BY stock_num
```

Query 3-11

manu_code	manu_code	manu_code	stock_num	description
HRO	HSK	SMT	1	baseball gloves
ANZ	NRG	SMT	5	tennis racquet
ANZ	HRO	HSK	110	helmet
ANZ	HRO	PRC	110	helmet
ANZ	HRO	SHM	110	helmet
ANZ	HSK	PRC	110	helmet
ANZ	HSK	SHM	110	helmet
ANZ	PRC	SHM	110	helmet
HRO	HSK	PRC	110	helmet
HRO	HSK	SHM	110	helmet
HRO	PRC	SHM	110	helmet
HSK	PRC	SHM	110	helmet
ANZ	KAR	NKL	201	golf shoes
ANZ	HRO	NKL	205	3 golf balls
ANZ	HRO	KAR	301	running shoes
.				
.				
.				
HRO	PRC	SHM	301	running shoes
KAR	NKL	PRC	301	running shoes
KAR	NKL	SHM	301	running shoes
KAR	PRC	SHM	301	running shoes
NKL	PRC	SHM	301	running shoes

Query Result 3-11

Say you want to select rows from a payroll table to determine which employees earn more than their manager. You can construct the self-join shown in Query 3-12a:

```
SELECT emp.employee_num, emp.gross_pay, emp.level,  
       emp.dept_num, mgr.employee_num, mgr.gross_pay,  
       mgr.dept_num, mgr.level  
FROM payroll emp, payroll mgr  
WHERE emp.gross_pay > mgr.gross_pay  
      AND emp.level < mgr.level  
      AND emp.dept_num = mgr.dept_num  
ORDER BY 4
```

Query 3-12a

Query 3-12b uses a *correlated subquery* to retrieve and list the 10 highest-priced items ordered:

```
SELECT order_num, total_price  
FROM items a  
WHERE 10 >  
      (SELECT COUNT (*)  
       FROM items b  
        WHERE b.total_price < a.total_price)  
ORDER BY total_price
```

Query 3-12b

Query 3-12b returns the 10 rows shown in Query Result 3-12:

order_num	total_price
1018	\$15.00
1013	\$19.80
1003	\$20.00
1005	\$36.00
1006	\$36.00
1013	\$36.00
1010	\$36.00
1013	\$40.00
1022	\$40.00
1023	\$40.00

Query Result 3-12

You can create a similar query to find and list the 10 employees in the company who have the most seniority.

Correlated and uncorrelated subqueries are described later in “Subqueries in SELECT Statements” on page 3-32.

Using Rowid Values

You can use the hidden *rowid* column in a self-join to locate duplicate values in a table. In the following example, the condition `x.rowid != y.rowid` is equivalent to saying “row x is not the same row as row y.”

Query 3-13 selects data twice from the **cust_calls** table, assigning it the table aliases **x** and **y**:

```
SELECT x.rowid, x.customer_num
       FROM cust_calls x, cust_calls y
       WHERE x.customer_num = y.customer_num
             AND x.rowid != y.rowid
```

Query 3-13

Query 3-13 searches for duplicate values in the **customer_num** column, and for their rowids, finding the pair shown in Query Result 3-13:

rowid	customer_num
515	116
769	116

Query Result 3-13

You can write the last condition in Query 3-13 as shown in the following example:

```
AND x.rowid != y.rowid

AND NOT x.rowid = y.rowid
```

Another way to locate duplicate values is with a correlated subquery, as shown in Query 3-14:

```
SELECT x.customer_num, x.call_dtime
FROM cust_calls x
WHERE 1 <
      (SELECT COUNT (*) FROM cust_calls y
       WHERE x.customer_num = y.customer_num)
```

Query 3-14

Query 3-14 locates the same two duplicate **customer_num** values as Query 3-13 and returns the rows shown in Query Result 3-14:

customer_num	call_dtime
116	1992-11-28 13:34
116	1992-12-21 11:24

Query Result 3-14

You can use the rowid, shown earlier in a self-join, to locate the internal record number associated with a row in a database table. The rowid is, in effect, a hidden column in every table. The sequential values of rowid have no special significance and may vary depending on the location of the physical data in the chunk. Your rowid may vary from the example shown. See Chapter 13, “Optimizing Your Queries” in this manual for a discussion of performance issues and the rowid value. The use of rowid is also discussed in detail in the *INFORMIX-OnLine Dynamic Server Administrator’s Guide*.

Query 3-16 uses rowid and the wildcard * in the SELECT clause to retrieve every row in the **manufact** table and their corresponding rowids:

```
SELECT rowid, * FROM manufact
```

Query 3-15

rowid	manu_code	manu_name	lead_time
257	SMT	Smith	3
258	ANZ	Anza	5
259	NRG	Norge	7
260	HSK	Husky	5
261	HRO	Hero	4
262	SHM	Shimara	30
263	KAR	Karsten	21
264	NKL	Nikolus	8
265	PRC	ProCycle	9

Query Result 3-15

You also can use rowid when you select a specific column, as shown in Query 3-16:

```
SELECT rowid, manu_code FROM manufact
```

Query 3-16

rowid	manu_code
258	ANZ
261	HRO
260	HSK
263	KAR
264	NKL
259	NRG
265	PRC
262	SHM
257	SMT

Query Result 3-16

You also can use the rowid in the WHERE clause to retrieve rows based on their internal record number. This method is handy when no other unique column exists in a table. Query 3-17 uses the rowid from Query 3-16:

```
SELECT * FROM manufact WHERE rowid = 263
```

Query 3-17

Query 3-17 returns the row shown in Query Result 3-17:

manu_code	manu_name	lead_time
KAR	Karsten	21

Query Result 3-17

Using the USER Function

To obtain additional information about a table, you can combine the rowid with the USER function.

Query 3-18 assigns the label **username** to the USER expression column and returns this information about the **cust_calls** table:

```
SELECT USER username, rowid FROM cust_calls
```

Query 3-18

username	rowid
zenda	257
zenda	258
zenda	259
zenda	513
zenda	514
zenda	515
zenda	769

Query Result 3-18

You also can use the USER function in a WHERE clause when you select the rowid.

Query 3-19 returns the rowid for only those rows inserted or updated by the user who performs the query:

```
SELECT rowid FROM cust_calls WHERE user_id = USER
```

Query 3-19

For example, if the user **richc** used Query 3-19, the output would be as shown in Query Result 3-19:

rowid
258
259

Query Result 3-19

Using the DBSERVERNAME Function

With **INFORMIX-OnLine Dynamic Server**, you can add the **DBSERVERNAME** keyword (or its synonym, **SITENAME**) to a query to find out where the current database resides.

Query 3-20 finds the database server name and the user name as well as the rowid and the *tabid*, which is the serial interval table identifier for system catalog tables:

```
SELECT DBSERVERNAME server, tabid, rowid, USER username
FROM systables
WHERE tabid >= 105 OR rowid <= 260
ORDER BY rowid
```

Query 3-20

Query 3-20 assigns display labels to the DBSERVERNAME and USER expressions and returns the 10 rows from the **systables** system catalog table, as shown in Query Result 3-20:

server	tabid	rowid	username
manatee	1	257	zenda
manatee	2	258	zenda
manatee	3	259	zenda
manatee	4	260	zenda
manatee	105	274	zenda
manatee	106	1025	zenda
manatee	107	1026	zenda
manatee	108	1027	zenda
manatee	109	1028	zenda
manatee	110	1029	zenda

Query Result 3-20

Note that you should never store a rowid in a *permanent* table or attempt to use it as a foreign key because the rowid can change. For example, if a table is dropped and then reloaded from external data, all the rowids are different.

USER and DBSERVERNAME were discussed in Chapter 2, “Composing Simple SELECT Statements.”

Outer Joins

Chapter 2, “Composing Simple SELECT Statements,” showed how to create and use some simple joins. Whereas a simple join treats two or more joined tables equally, an *outer join* treats two or more joined tables *unsymmetrically*. It makes one of the tables *dominant* (also called “preserved”) over the other *subservient* tables.

Outer joins occur in four basic types:

- A simple outer join on two tables
- A simple outer join to a third table
- An outer join for a simple join to a third table
- An outer join for an outer join to a third table

This section discusses these types of outer joins. See the discussion of outer joins in Chapter 1 of the *Informix Guide to SQL: Syntax* for full information on their syntax, use, and logic.

In a *simple join*, the result contains only the combinations of rows from the tables that satisfy the join conditions. *Rows that do not satisfy the join conditions are discarded.*

In an *outer join*, the result contains the combinations of rows from the tables that satisfy the join conditions. *Rows from the dominant table that would otherwise be discarded are preserved, even though no matching row was found in the subservient table.* The dominant-table rows that do not have a matching subservient-table row receive a row of nulls before the selected columns are projected.

An outer join applies conditions to the subservient table while sequentially applying the join conditions to the rows of the dominant table. The conditions are expressed in a WHERE clause.

An outer join must have a SELECT clause, a FROM clause, and a WHERE clause. You transform a simple join into an outer join by inserting the keyword OUTER directly before the name of the subservient tables in the FROM clause. As shown later in this section, you can include the OUTER keyword more than once in your query.

Before you use outer joins heavily, you should determine whether one or more simple joins can work. You often can get by with a simple join when you do not need supplemental information from other tables.

The examples in this section use table aliases for brevity. Table aliases are discussed in the preceding chapter.

Simple Join

Query 3-21 is an example of the type of simple join on the **customer** and **cust_calls** tables shown in Chapter 2, “Composing Simple SELECT Statements.”

```
SELECT c.customer_num, c.lname, c.company,
       c.phone, u.call_dtime, u.call_descr
FROM customer c, cust_calls u
WHERE c.customer_num = u.customer_num
```

Query 3-21

Query 3-21 returns only those rows where the customer has made a call to customer service, as shown in Query Result 3-21:

```
customer_num 106
lname        Watson
company      Watson & Son
phone        415-389-8789
call_dtime   1993-06-12 08:20
call_descr   Order was received, but two of the cans of
              ANZ tennis balls within the case were empty

customer_num 110
lname        Jaeger
company      AA Athletics
phone        415-743-3611
call_dtime   1993-07-07 10:24
call_descr   Order placed one month ago (6/7) not received.

customer_num 119
lname        Shorter
company      The Triathletes Club
phone        609-663-6079
call_dtime   1993-07-01 15:00
call_descr   Bill does not reflect credit from previous order

customer_num 121
lname        Wallack
company      City Sports
phone        302-366-7511
call_dtime   1993-07-10 14:05
call_descr   Customer likes our merchandise. Requests that we
              stock more types of infant joggers. Will call back
              to place order.

customer_num 127
lname        Satifer
company      Big Blue Bike Shop
phone        312-944-5691
call_dtime   1993-07-31 14:30
call_descr   Received Hero watches (item # 304) instead of
              ANZ watches

customer_num 116
lname        Parmelee
company      Olympic City
phone        415-534-8822
call_dtime   1992-11-28 13:34
call_descr   Received plain white swim caps (313 ANZ) instead
              of navy with team logo (313 SHM)

customer_num 116
lname        Parmelee
company      Olympic City
phone        415-534-8822
call_dtime   1992-12-21 11:24
call_descr   Second complaint from this customer! Received
              two cases right-handed outfielder gloves (1 HRO)
              instead of one case lefties.
```

Query Result 3-21

Simple Outer Join on Two Tables

Query 3-22 uses the same select list, tables, and comparison condition as the preceding example, but this time it creates a simple outer join:

```
SELECT c.customer_num, c.lname, c.company,  
       c.phone, u.call_dtime, u.call_descr  
FROM customer c, OUTER cust_calls u  
WHERE c.customer_num = u.customer_num
```

Query 3-22

The addition of the keyword **OUTER** in front of the **cust_calls** table makes it the subservient table. An outer join causes the query to return information on *all* customers, whether or not they have made calls to customer service. All rows from the dominant **customer** table are retrieved, and null values are assigned to corresponding rows from the subservient **cust_calls** table.


```

customer_num 101
lname       Pauli
company     All Sports Supplies
phone       408-789-8075
call_dtime
call_descr

customer_num 102
lname       Sadler
company     Sports Spot
phone       415-822-1289
call_dtime
call_descr

customer_num 103
lname       Currie
company     Phil's Sports
phone       415-328-4543
call_dtime
call_descr

customer_num 104
lname       Higgins
company     Play Ball!
phone       415-368-1100
call_dtime
call_descr

customer_num 105
lname       Vector
company     Los Altos Sports
phone       415-776-3249
call_dtime
call_descr

customer_num 106
lname       Watson
company     Watson & Son
phone       415-389-8789
call_dtime  1993-06-12 08:20
call_descr  Order was received, but two of the cans of
              ANZ tennis balls within the case were empty

customer_num 107
lname       Ream
company     Athletic Supplies
phone       415-356-9876
call_dtime
call_descr

customer_num 108
lname       Quinn
company     Quinn's Sports
phone       415-544-8729
call_dtime
call_descr
.
.
.

```

Query Result 3-22

Outer Join for a Simple Join to a Third Table

Query 3-23 shows an outer join that is the result of a simple join to a third table. This second type of outer join is known as a *nested simple join*.

```
SELECT c.customer_num, c.lname, o.order_num,  
       i.stock_num, i.manu_code, i.quantity  
FROM customer c, OUTER (orders o, items i)  
WHERE c.customer_num = o.customer_num  
      AND o.order_num = i.order_num  
      AND manu_code IN ('KAR', 'SHM')  
ORDER BY lname
```

Query 3-23

Query 3-23 first performs a simple join on the **orders** and **items** tables, retrieving information on all orders for items with a **manu_code** of KAR or SHM. It then performs an outer join to combine this information with data from the dominant **customer** table. An optional ORDER BY clause reorganizes the data into the form shown in Query Result 3-23:

customer_num	lname	order_num	stock_num	manu_code	quantity
114	Albertson				
118	Baxter				
113	Beatty				
103	Currie				
115	Grant				
123	Hanlon	1020	301	KAR	4
123	Hanlon	1020	204	KAR	2
125	Henry				
104	Higgins				
110	Jaeger				
120	Jewell	1017	202	KAR	1
120	Jewell	1017	301	SHM	2
111	Keyes				
112	Lawson				
128	Lessor				
109	Miller				
126	Neelie				
122	O'Brian	1019	111	SHM	3
116	Parmelee				
101	Pauli				
124	Putnum	1021	202	KAR	3
108	Quinn				
107	Ream				
102	Sadler				
127	Satifer	1023	306	SHM	1
127	Satifer	1023	105	SHM	1
127	Satifer	1023	110	SHM	1
119	Shorter	1016	101	SHM	2
117	Sipes				
105	Vector				
121	Wallack	1018	302	KAR	3
106	Watson				

Query Result 3-23

Outer Join for an Outer Join to a Third Table

Query 3-24 creates an outer join that is the result of an outer join to a third table. This third type is known as a *nested outer join*.

```
SELECT c.customer_num, lname, o.order_num,
       stock_num, manu_code, quantity
FROM customer c, OUTER (orders o, OUTER items i)
WHERE c.customer_num = o.customer_num
      AND o.order_num = i.order_num
      AND manu_code IN ('KAR', 'SHM')
ORDER BY lname
```

Query 3-24

Query 3-24 first performs an outer join on the **orders** and **items** tables, retrieving information on all orders for items with a **manu_code** of KAR or SHM. It then performs an outer join, which combines this information with data from the dominant **customer** table. Query 3-24 preserves order numbers

that the previous example eliminated, returning rows for orders that do not contain items with either manufacturer code. An optional ORDER BY clause reorganizes the data, as shown in Query Result 3-24:

customer_num	lname	order_num	stock_num	manu_code	quantity
114	Albertson				
118	Baxter				
113	Beatty				
103	Currie				
115	Grant	1010			
123	Hanlon	1020	204	KAR	2
123	Hanlon	1020	301	KAR	4
125	Henry				
104	Higgins	1011			
104	Higgins	1001			
104	Higgins	1013			
104	Higgins	1003			
110	Jaeger	1008			
110	Jaeger	1015			
120	Jewell	1017	301	SHM	2
120	Jewell	1017	202	KAR	1
111	Keyes	1009			
112	Lawson	1006			
128	Lessor				
109	Miller				
126	Neelie	1022			
122	O'Brian	1019	111	SHM	3
116	Parmelee	1005			
101	Pauli	1002			
124	Putnum	1021	202	KAR	3
108	Quinn				
107	Ream				
102	Sadler				
127	Satifer	1023	110	SHM	1
127	Satifer	1023	105	SHM	1
127	Satifer	1023	306	SHM	1
119	Shorter	1016	101	SHM	2
117	Sipes	1012			
117	Sipes	1007			
105	Vector				
121	Wallack	1018	302	KAR	3
106	Watson	1014			
106	Watson	1004			

Query Result 3-24

You can state the join conditions in two ways when you apply an outer join to the result of an outer join to a third table. The two subservient tables are joined, but you can join the dominant table to either subservient table without affecting the results if the dominant table and the subservient table share a common column.

Outer Join of Two Tables to a Third Table

Query 3-25 shows an outer join that is the result of an outer join of each of two tables to a third table. In this fourth type of outer join, join relationships are possible *only* between the dominant table and the subservient tables.

```
SELECT c.customer_num, lname, o.order_num,
       order_date, call_dtime
FROM customer c, OUTER orders o, OUTER cust_calls x
WHERE c.customer_num = o.customer_num
      AND c.customer_num = x.customer_num
ORDER BY lname
INTO TEMP service
```

Query 3-25

Query 3-25 individually joins the subservient tables **orders** and **cust_calls** to the dominant **customer** table; it does not join the two subservient tables. An INTO TEMP clause selects the results into a temporary table for further manipulation or queries, as shown in Query Result 3-25.

customer_num	lname	order_num	order_date	call_dtime
114	Albertson			
118	Baxter			
113	Beatty			
103	Currie			
115	Grant	1010	06/17/1993	
123	Hanlon	1020	07/11/1993	
125	Henry			
104	Higgins	1003	05/22/1993	
104	Higgins	1001	05/20/1993	
104	Higgins	1013	06/22/1993	
104	Higgins	1011	06/18/1993	
110	Jaeger	1015	06/27/1993	1993-07-07 10:24
110	Jaeger	1008	06/07/1993	1993-07-07 10:24
120	Jewell	1017	07/09/1993	
111	Keyes	1009	06/14/1993	
112	Lawson	1006	05/30/1993	
109	Miller			
128	Moore			
126	Neelie	1022	07/24/1993	
122	O'Brian	1019	07/11/1993	
116	Parmelee	1005	05/24/1993	1992-12-21 11:24
116	Parmelee	1005	05/24/1993	1992-11-28 13:34
101	Pauli	1002	05/21/1993	
124	Putnum	1021	07/23/1993	
108	Quinn			
107	Ream			
102	Sadler			
127	Satifer	1023	07/24/1993	1993-07-31 14:30
119	Shorter	1016	06/29/1993	1993-07-01 15:00
117	Sipes	1007	05/31/1993	
117	Sipes	1012	06/18/1993	
105	Vector			
121	Wallack	1018	07/10/1993	1993-07-10 14:05
106	Watson	1004	05/22/1993	1993-06-12 08:20
106	Watson	1014	06/25/1993	1993-06-12 08:20

Query Result 3-25

Note that if Query 3-25 had tried to create a join condition between the two subservient tables **o** and **x**, as shown in Query 3-26, an error message would have indicated the creation of a two-sided outer join:

```
WHERE o.customer_num = x.customer_num
```

Query 3-26

Subqueries in SELECT Statements

A SELECT statement *nested* in the WHERE clause of another SELECT statement (or in an INSERT, DELETE, or UPDATE statement) is called a *subquery*. Each subquery must contain a SELECT clause and a FROM clause, and it must be enclosed in parentheses, which tells the database server to perform that operation first.

Subqueries can be *correlated* or *uncorrelated*. A subquery (or *inner* SELECT statement) is correlated when the value it produces depends on a value produced by the *outer* SELECT statement that contains it. Any other kind of subquery is considered uncorrelated.

The important feature of a correlated subquery is that, because it depends on a value from the outer SELECT, it must be executed repeatedly, once for every value produced by the outer SELECT. An uncorrelated subquery is executed only once.

You can construct a SELECT statement with a subquery to replace two separate SELECT statements.

Subqueries in SELECT statements allow you to perform the following actions:

- Compare an expression to the result of another SELECT statement.
- Determine whether an expression is included in the results of another SELECT statement.
- Determine whether any rows are selected by another SELECT statement.

An optional WHERE clause in a subquery often is used to narrow the search condition.

A subquery selects and returns values to the first or outer SELECT statement. A subquery can return no value, a single value, or a set of values.

- If it returns *no* value, the query does not return any rows. Such a subquery is equivalent to a null value.
- If it returns *one* value, the subquery returns either one aggregate expression or else selects exactly one row and one column. Such a subquery is equivalent to a single number or character value.
- If it returns a list or *set* of values, the subquery returns either one row or one column.

The following keywords introduce a subquery in the WHERE clause of a SELECT statement:

- ALL
- ANY
- IN
- EXISTS

You can use any of the relational operators with ALL and ANY to compare something to every one of (ALL), or to any one of (ANY), the values that the subquery produces. You can use the keyword SOME in place of ANY. The operator IN is equivalent to =ANY. To create the opposite search condition, use the keyword NOT or a different relational operator.

The EXISTS operator tests a subquery to see if it found any values; that is, it asks if the result of the subquery is not null.

See Chapter 1 in the *Informix Guide to SQL: Syntax* for the complete syntax used in creating a condition with a subquery. See also Chapter 13, “Optimizing Your Queries,” of this manual for information on performance implications for correlated and uncorrelated subqueries.

Using ALL

Use the keyword ALL preceding a subquery to determine whether a comparison is true for every value returned. If the subquery returns no values, the search condition is *true*. (If it returns no values, the condition is true of all of the zero values.)

Query 3-27 lists the following information for all orders that contain an item for which the total price is less than the total price on *every* item in order number 1023:

```
SELECT order_num, stock_num, manu_code, total_price
FROM items
WHERE total_price < ALL
      (SELECT total_price FROM items
       WHERE order_num = 1023)
```

Query 3-27

order_num	stock_num	manu_code	total_price
1003	9	ANZ	\$20.00
1005	6	SMT	\$36.00
1006	6	SMT	\$36.00
1010	6	SMT	\$36.00
1013	5	ANZ	\$19.80
1013	6	SMT	\$36.00
1018	302	KAR	\$15.00

Query Result 3-27

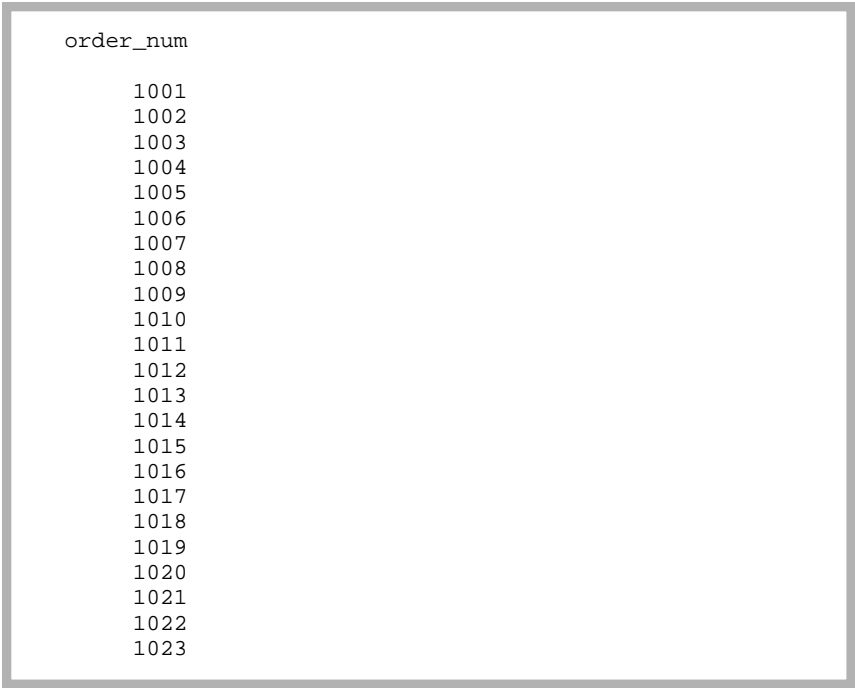
Using ANY

Use the keyword ANY (or its synonym SOME) preceding a subquery to determine whether a comparison is true for at least one of the values returned. If the subquery returns no values, the search condition is *false*. (Because there were no values, the condition cannot be true for one of them.)

Query 3-28 finds the order number of all orders that contain an item for which the total price is greater than the total price of *any one* of the items in order number 1005.

```
SELECT DISTINCT order_num
  FROM items
 WHERE total_price > ANY
    (SELECT total_price
      FROM items
     WHERE order_num = 1005)
```

Query 3-28



order_num

1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023

Query Result 3-28

Single-Valued Subqueries

You do not need to include the keyword ALL or ANY if you know the subquery can return *exactly one value* to the outer-level query. A subquery that returns exactly one value can be treated like a function. This kind of subquery often uses an aggregate function because aggregate functions always return single values

Query 3-29 uses the aggregate function MAX in a subquery to find the **order_num** for orders that include the maximum number of volleyball nets.

```
SELECT order_num FROM items
WHERE stock_num = 9
AND quantity =
    (SELECT MAX (quantity)
     FROM items
     WHERE stock_num = 9)
```

Query 3-29

order_num
1012

Query Result 3-29

Query 3-30 uses the aggregate function MIN in the subquery to select items for which the total price is higher than 10 times the minimum price:

```
SELECT order_num, stock_num, manu_code, total_price
FROM items x
WHERE total_price >
      (SELECT 10 * MIN (total_price)
       FROM items
       WHERE order_num = x.order_num)
```

Query 3-30

order_num	stock_num	manu_code	total_price
1003	8	ANZ	\$840.00
1018	307	PRC	\$500.00
1018	110	PRC	\$236.00
1018	304	HRO	\$280.00

Query Result 3-30

Correlated Subqueries

Query 3-31 is an example of a correlated subquery, which returns a list of the 10 earliest shipping dates in the **orders** table. It includes an ORDER BY clause after the subquery to order the results because you cannot include ORDER BY within a subquery.

```
SELECT po_num, ship_date FROM orders main
WHERE 10 >
      (SELECT COUNT (DISTINCT ship_date)
       FROM orders sub
       WHERE sub.ship_date > main.ship_date)
       AND ship_date IS NOT NULL
ORDER BY ship_date, po_num
```

Query 3-31

The subquery is correlated because the number it produces depends on **main.ship_date**, a value produced by the outer SELECT. Thus, the subquery must be executed anew for every row that the outer query considers.

Query 3-31 uses the COUNT function to return a value to the main query. The ORDER BY clause then orders the data. The query locates and returns the 13 rows that have the 10 latest shipping dates, as shown in Query Result 3-31:

po_num	ship_date
4745	06/21/1993
278701	06/29/1993
429Q	06/29/1993
8052	07/03/1993
B77897	07/03/1993
LZ230	07/06/1993
B77930	07/10/1993
PC6782	07/12/1993
DM354331	07/13/1993
S22942	07/13/1993
MA003	07/16/1993
W2286	07/16/1993
Z55709	07/16/1993
C3288	07/25/1993
KF2961	07/30/1993
W9925	07/30/1993

Query Result 3-31

If you use a correlated subquery, such as Query 3-31, on a very large table, you should index the **ship_date** column to improve performance. Otherwise, this SELECT statement might be considered somewhat inefficient because it executes the subquery once for every row of the table. Indexing and performance issues are discussed in Chapter 10, “Tuning Your Data Model.”

Using EXISTS

The keyword EXISTS is known as an *existential qualifier* because the subquery is true only if the outer SELECT, shown in Query 3-32a, finds at least one row:

```
SELECT UNIQUE manu_name, lead_time
  FROM manufact
 WHERE EXISTS
    (SELECT * FROM stock
     WHERE description MATCHES '*shoe*'
        AND manufact.manu_code = stock.manu_code)
```

Query 3-32a

You often can construct a query with EXISTS that is equivalent to one that uses IN. You also can substitute =ANY for IN, as shown in Query 3-32b:

```
SELECT UNIQUE manu_name, lead_time
  FROM stock, manufact
 WHERE manufact.manu_code IN
    (SELECT manu_code FROM stock
     WHERE description MATCHES '*shoe*')
    AND stock.manu_code = manufact.manu_code
```

Query 3-32b

Query 3-32a and Query 3-32b return rows for the manufacturers that produce a kind of shoe as well as the lead time for ordering the product. These results are shown in Query Result 3-32:

manu_name	lead_time
Anza	5
Hero	4
Karsten	21
Nikolus	8
ProCycle	9
Shimara	30

Query Result 3-32

Note that you cannot use the predicate IN for a subquery that contains a column with a TEXT or BYTE data type.

Add the keyword NOT to IN or to EXISTS to create a search condition that is the opposite of the one in the preceding queries. You also can substitute !=ALL for NOT IN.

Query 3-33 shows two ways to do the same thing. One way might allow the database server to do less work than the other, depending on the design of the database and the size of the tables. To find out which query might be

better, you can use the SET EXPLAIN command to get a listing of the query plan. SET EXPLAIN is discussed in Chapter 13, “Optimizing Your Queries,” of this manual and in Chapter 1 of the *Informix Guide to SQL: Syntax*.

```
SELECT customer_num, company FROM customer
  WHERE customer_num NOT IN
    (SELECT customer_num FROM orders
     WHERE customer.customer_num = orders.customer_num)

SELECT customer_num, company FROM customer
  WHERE NOT EXISTS
    (SELECT * FROM orders
     WHERE customer.customer_num = orders.customer_num)
```

Query 3-33

Each statement in Query 3-33 returns the rows shown in Query Result 3-33, which identify customers who have not placed orders:

```
customer_num company
102 Sports Spot
103 Phil's Sports
105 Los Altos Sports
107 Athletic Supplies
108 Quinn's Sports
109 Sport Stuff
113 Sportstown
114 Sporting Place
118 Blue Ribbon Sports
125 Total Fitness Sports
128 Phoenix University
```

Query Result 3-33

Note that the keywords EXISTS and IN are used for the set operation known as *intersection*, and the keywords NOT EXISTS and NOT IN are used for the set operation known as *difference*. These concepts are discussed in “Set Operations” on page 3-42.

Query 3-34 identifies all the items in the **stock** table that have not been ordered yet by performing a subquery on the **items** table:

```
SELECT stock.* FROM stock
WHERE NOT EXISTS
  (SELECT * FROM items
   WHERE stock.stock_num = items.stock_num
     AND stock.manu_code = items.manu_code)
```

Query 3-34

It returns the rows shown in Query Result 3-34:

stock_num	manu_code	description	unit_price	unit	unit_descr
101	PRC	bicycle tires	\$88.00	box	4/box
102	SHM	bicycle brakes	\$220.00	case	4 sets/case
102	PRC	bicycle brakes	\$480.00	case	4 sets/case
105	PRC	bicycle wheels	\$53.00	pair	pair
106	PRC	bicycle stem	\$23.00	each	each
107	PRC	bicycle saddle	\$70.00	pair	pair
108	SHM	crankset	\$45.00	each	each
109	SHM	pedal binding	\$200.00	case	4 pairs/case
110	ANZ	helmet	\$244.00	case	4/case
110	HRO	helmet	\$260.00	case	4/case
112	SHM	12-spd, assmbld	\$549.00	each	each
113	SHM	18-spd, assmbld	\$685.90	each	each
201	KAR	golf shoes	\$90.00	each	each
202	NKL	metal woods	\$174.00	case	2 sets/case
203	NKL	irons/wedge	\$670.00	case	2 sets/case
205	NKL	3 golf balls	\$312.00	case	24/case
205	HRO	3 golf balls	\$312.00	case	24/case
301	NKL	running shoes	\$97.00	each	each
301	HRO	running shoes	\$42.50	each	each
301	PRC	running shoes	\$75.00	each	each
301	ANZ	running shoes	\$95.00	each	each
302	HRO	ice pack	\$4.50	each	each
303	KAR	socks	\$36.00	box	24 pairs/box
305	HRO	first-aid kit	\$48.00	case	4/case
306	PRC	tandem adapter	\$160.00	each	each
308	PRC	twin jogger	\$280.00	each	each
309	SHM	ear drops	\$40.00	case	20/case
310	SHM	kick board	\$80.00	case	10/case
310	ANZ	kick board	\$84.00	case	12/case
311	SHM	water gloves	\$48.00	box	4 pairs/box
312	SHM	racer goggles	\$96.00	box	12/box
312	HRO	racer goggles	\$72.00	box	12/box
313	SHM	swim cap	\$72.00	box	12/box
313	ANZ	swim cap	\$60.00	box	12/box

Query Result 3-34

Note that no logical limit exists to the number of subqueries a SELECT statement can have, but the size of any statement is physically limited when considered as a character string. However, this limit is probably larger than any practical statement you are likely to compose.

Perhaps you want to check whether information has been entered correctly in the database. One way to find errors in a database is to write a query that returns output only when errors exist. A subquery of this type serves as a kind of *audit query*, as shown in Query 3-35:

```
SELECT * FROM items
      WHERE total_price != quantity *
            (SELECT unit_price FROM stock
              WHERE stock.stock_num = items.stock_num
                 AND stock.manu_code = items.manu_code)
```

Query 3-35

Query Result 3-35 returns only those rows for which the total price of an item on an order is not equal to the stock unit price times the order quantity. Assuming that no discount has been applied, such rows must have been entered incorrectly in the database.

item_num	order_num	stock_num	manu_code	quantity	total_price
1	1004	1	HRO	1	\$960.00
2	1006	5	NRG	5	\$190.00

Query Result 3-35

Query 3-35 returns rows only when errors occur. If information is correctly inserted into the database, no rows are returned.

Set Operations

The standard set operations *union*, *intersection*, and *difference* let you manipulate database information. These three operations enable you to use SELECT statements to check the integrity of your database after you perform an update, insert, or delete. They can be useful when you transfer data to a history table, for example, and want to verify that the correct data is in the history table before you delete it from the original table.

Union

The union operation uses the UNION keyword, or *operator*, to combine two queries into a single *compound query*. You can use the UNION keyword between two or more SELECT statements to *unite* them and produce a temporary table containing rows that exist in any or all of the original tables. (Note that you cannot use a UNION operator inside a subquery or in the definition of a view.) Figure 3-1 illustrates the union set operation.

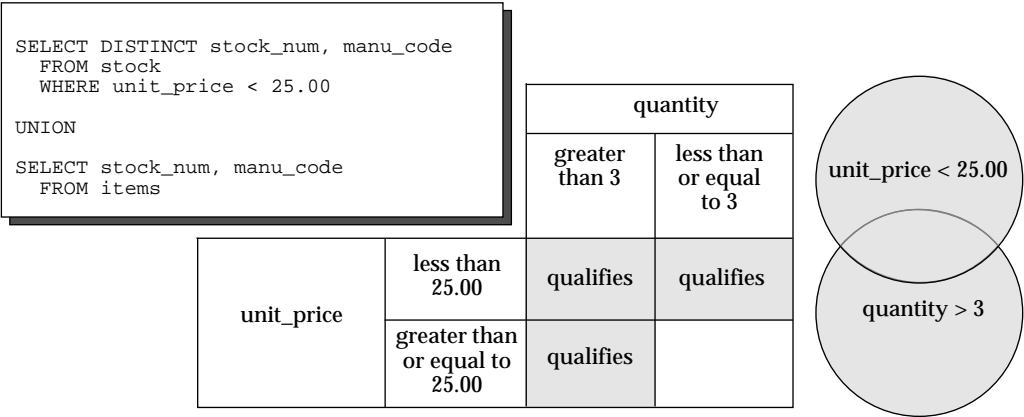


Figure 3-1 *The union set operation*

The UNION keyword selects all rows from the two queries, removes duplicates, and returns what is left. Because the results of the queries are combined into a single result, the select list in each query must have the same number of columns. Also, the corresponding columns selected from each table must be of the same data type (CHARACTER type columns must be the same length), and these corresponding columns must either all allow or all disallow nulls.

Query 3-36 performs a union on the **stock_num** and **manu_code** columns in the **stock** and **items** tables:

```
SELECT DISTINCT stock_num, manu_code
  FROM stock
 WHERE unit_price < 25.00

UNION

SELECT stock_num, manu_code
  FROM items
 WHERE quantity > 3
```

Query 3-36

Query 3-36 selects those items that have a unit price of less than \$25.00 or that have been ordered in quantities greater than three and lists their **stock_num** and **manu_code**, as shown in Query Result 3-36:

stock_num	manu_code
5	ANZ
5	NRG
5	SMT
9	ANZ
103	PRC
106	PRC
201	NKL
301	KAR
302	HRO
302	KAR

Query Result 3-36

If you include an ORDER BY clause, it must follow Query 3-36 and use an integer, not an identifier, to refer to the ordering column. Ordering takes place after the set operation is complete.

```
SELECT DISTINCT stock_num, manu_code
  FROM stock
 WHERE unit_price < 25.00

UNION

SELECT stock_num, manu_code
  FROM items
 WHERE quantity > 3
 ORDER BY 2
```

Query 3-37

The compound query in Query 3-37 selects the same rows as Query 3-36 but displays them in order of manufacturer code, as shown in Query Result 3-37:

stock_num	manu_code
5	ANZ
9	ANZ
302	HRO
301	KAR
302	KAR
201	NKL
5	NRG
103	PRC
106	PRC
5	SMT

Query Result 3-37

By default, the UNION keyword excludes duplicate rows. Add the optional keyword ALL, as shown in Query 3-38, to retain the duplicate values:

```
SELECT stock_num, manu_code
      FROM stock
      WHERE unit_price < 25.00

UNION ALL

SELECT stock_num, manu_code
      FROM items
      WHERE quantity > 3
      ORDER BY 2
      INTO TEMP stockitem
```

Query 3-38

Query 3-38 uses the UNION ALL keywords to unite two SELECT statements and puts the results into a temporary table by adding an INTO TEMP clause after the final SELECT. It returns the same rows as Query 3-37 but also includes duplicate values.

stock_num	manu_code
9	ANZ
5	ANZ
9	ANZ
5	ANZ
9	ANZ
5	ANZ
5	ANZ
5	ANZ
302	HRO
302	KAR
301	KAR
201	NKL
5	NRG
5	NRG
103	PRC
106	PRC
5	SMT
5	SMT

Query Result 3-38

Corresponding columns in the select lists for the combined queries must have identical data types, but the columns do not need to use the same identifier.

Query 3-39 selects the **state** column from the **customer** table and the corresponding **code** column from the **state** table:

```
SELECT DISTINCT state
  FROM customer
 WHERE customer_num BETWEEN 120 AND 125

UNION

SELECT DISTINCT code
  FROM state
 WHERE sname MATCHES '*a'
```

Query 3-39

Query Result 3-39 returns state code abbreviations for customer numbers 120 through 125, or for states whose **sname** ends in A or a:

```
state

AK
AL
AZ
CA
DE
FL
GA
IA
IN
LA
MA
MN
MT
NC
ND
NE
NJ
NV
OK
PA
SC
SD
VA
WV
```

Query Result 3-39

In compound queries, the column names or display labels in the first SELECT statement are the ones that appear in the results. Thus, in Query 3-40, the column name **state** from the first SELECT statement is used instead of the column name **code** from the second.

Query 3-40 performs a union on three tables. The maximum number of unions depends on the practicality of the application and any memory limitations.

```
SELECT stock_num, manu_code
      FROM stock
      WHERE unit_price > 600.00

UNION ALL

SELECT stock_num, manu_code
      FROM catalog
      WHERE catalog_num = 10025

UNION ALL

SELECT stock_num, manu_code
      FROM items
      WHERE quantity = 10
      ORDER BY 2
```

Query 3-40

Query 3-40 selects items where the **unit_price** in the **stock** table is greater than \$600, or the **catalog_num** in the **catalog** table is 10025, or the **quantity** in the **items** table is 10, and orders it by **manu_code**. The results are shown in Query Result 3-40:

stock_num	manu_code
5	ANZ
9	ANZ
8	ANZ
4	HSK
1	HSK
203	NKL
5	NRG
106	PRC
113	SHM

Query Result 3-40

See Chapter 1 of the *Informix Guide to SQL: Syntax* for the complete syntax of the SELECT statement and the UNION operator. See also Chapter 5, “Programming with SQL,” and Chapter 6, “Modifying Data Through SQL Programs,” as well as the product manuals for information specific to the **INFORMIX-4GL** and **INFORMIX-ESQL/C** products and any limitations involving the INTO clause and compound queries.

Query 3-41 uses a combined query to select data into a temporary table and then adds a simple query to order and display it. You must separate the combined and simple queries with a semicolon.

The combined query uses a literal in the select list to tag the output of part of a union so it can be distinguished later. The tag is given the label **sortkey**. The simple query uses that tag as a sort key for ordering the retrieved rows.

```
SELECT '1' sortkey, lname, fname, company,
       city, state, phone
FROM customer x
WHERE state = 'CA'

UNION

SELECT '2' sortkey, lname, fname, company,
       city, state, phone
FROM customer y
WHERE state <> 'CA'
INTO TEMP calcust;

SELECT * FROM calcust
ORDER BY 1
```

Query 3-41

Query 3-41 creates a list where the California customers, the ones called most frequently, appear first, as shown in Query Result 3-41:

```
sortkey 1
lname   Baxter
fname   Dick
company Blue Ribbon Sports
city     Oakland
state    CA
phone    415-655-0011

sortkey 1
lname   Beatty
fname   Lana
company Sportstown
city     Menlo Park
state    CA
phone    415-356-9982

sortkey 1
lname   Currie
fname   Philip
company Phil's Sports
city     Palo Alto
state    CA
phone    415-328-4543

sortkey 1
lname   Grant
fname   Alfred
company Gold Medal Sports
city     Menlo Park
state    CA
phone    415-356-1123
.
.
.
sortkey 2
lname   Satifer
fname   Kim
company Big Blue Bike Shop
city     Blue Island
state    NY
phone    312-944-5691

sortkey 2
lname   Shorter
fname   Bob
company The Triathletes Club
city     Cherry Hill
state    NJ
phone    609-663-6079

sortkey 2
lname   Wallack
fname   Jason
company City Sports
city     Wilmington
state    DE
phone    302-366-7511
```

Query Result 3-41

Intersection

The *intersection* of two sets of rows produces a table containing rows that exist in both of the original tables. Use the keyword EXISTS or IN to introduce sub-queries that show the intersection of two sets. Figure 3-2 illustrates the intersection set operation.

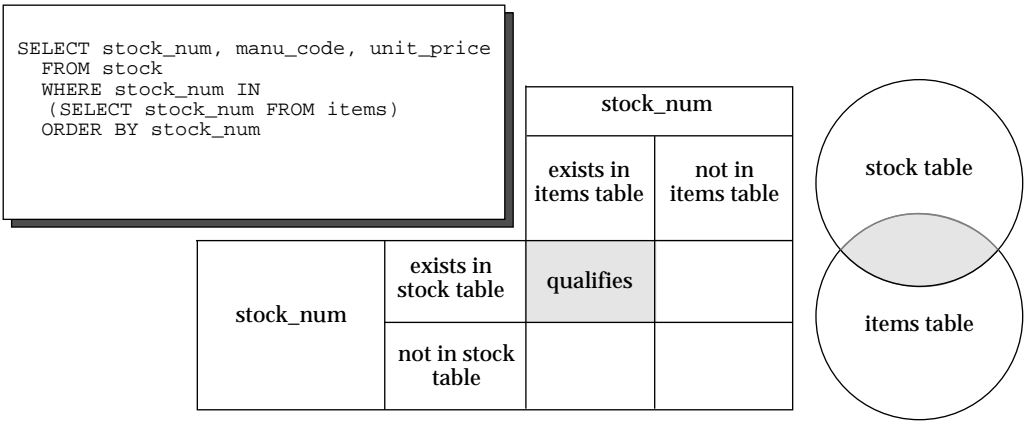


Figure 3-2 *The intersection set operation*

Query 3-42 is an example of a nested SELECT statement that shows the intersection of the **stock** and **items** tables:

```
SELECT stock_num, manu_code, unit_price
FROM stock
WHERE stock_num IN
  (SELECT stock_num FROM items)
ORDER BY stock_num
```

Query 3-42

Query Result 3-42 contains all the elements from both sets, returning the following 57 rows:

stock_num	manu_code	unit_price
1	HRO	\$250.00
1	HSK	\$800.00
1	SMT	\$450.00
2	HRO	\$126.00
3	HSK	\$240.00
3	SHM	\$280.00
4	HRO	\$480.00
4	HSK	\$960.00
5	ANZ	\$19.80
5	NRG	\$28.00
5	SMT	\$25.00
6	ANZ	\$48.00
6	SMT	\$36.00
7	HRO	\$600.00
8	ANZ	\$840.00
9	ANZ	\$20.00
101	PRC	\$88.00
101	SHM	\$68.00
103	PRC	\$20.00
104	PRC	\$58.00
105	PRC	\$53.00
105	SHM	\$80.00
109	PRC	\$30.00
109	SHM	\$200.00
110	ANZ	\$244.00
110	HRO	\$260.00
110	HSK	\$308.00
110	PRC	\$236.00
110	SHM	\$228.00
111	SHM	\$499.99
114	PRC	\$120.00
201	ANZ	\$75.00
201	KAR	\$90.00
201	NKL	\$37.50
202	KAR	\$230.00
202	NKL	\$174.00
204	KAR	\$45.00
205	ANZ	\$312.00
205	HRO	\$312.00
205	NKL	\$312.00
301	ANZ	\$95.00
301	HRO	\$42.50
301	KAR	\$87.00
301	NKL	\$97.00
301	PRC	\$75.00
301	SHM	\$102.00
302	HRO	\$4.50
302	KAR	\$5.00
303	KAR	\$36.00
303	PRC	\$48.00
304	ANZ	\$170.00
304	HRO	\$280.00
306	PRC	\$160.00
306	SHM	\$190.00
307	PRC	\$250.00
309	HRO	\$40.00
309	SHM	\$40.00

Query Result 3-42

Difference

The *difference* between two sets of rows produces a table containing rows in the first set that are not also in the second set. Use the keywords NOT EXISTS or NOT IN to introduce subqueries that show the difference between two sets. Figure 3-3 illustrates the difference set operation.

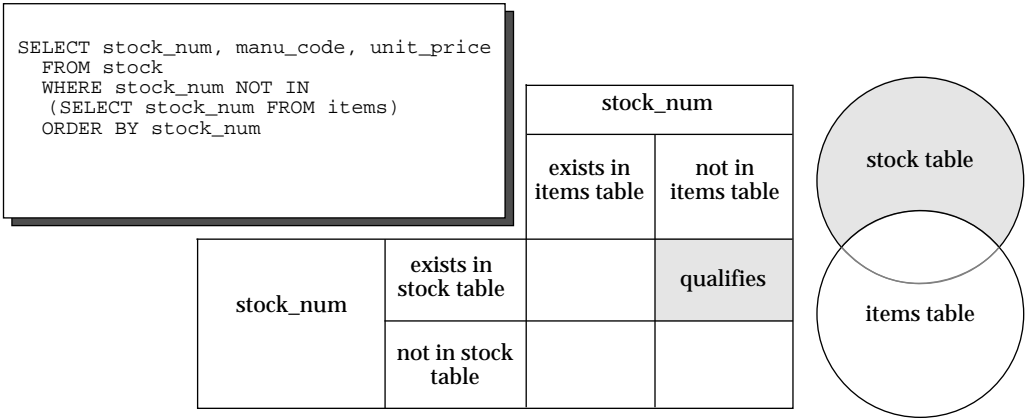


Figure 3-3 *The difference set operation*

Query 3-43 is an example of a nested SELECT statement that shows the difference between the **stock** and **items** tables:

```
SELECT stock_num, manu_code, unit_price
FROM stock
WHERE stock_num NOT IN
      (SELECT stock_num FROM items)
ORDER BY stock_num
```

Query 3-43

Query Result 3-43 contains all the elements from only the first set, returning the following 17 rows:

stock_num	manu_code	unit_price
102	PRC	\$480.00
102	SHM	\$220.00
106	PRC	\$23.00
107	PRC	\$70.00
108	SHM	\$45.00
112	SHM	\$549.00
113	SHM	\$685.90
203	NKL	\$670.00
305	HRO	\$48.00
308	PRC	\$280.00
310	ANZ	\$84.00
310	SHM	\$80.00
311	SHM	\$48.00
312	HRO	\$72.00
312	SHM	\$96.00
313	ANZ	\$60.00
313	SHM	\$72.00

Query Result 3-43

Summary

This chapter was built on concepts introduced in Chapter 2, “Composing Simple SELECT Statements.” It provided sample syntax and results for more advanced kinds of SELECT statements, which are used to perform a query on a relational database. This chapter

- Introduced the GROUP BY and HAVING clauses, which can be used with aggregates to return groups of rows and apply conditions to those groups.
- Described how to use the rowid to retrieve internal record numbers from tables and system catalog tables and discussed the serial internal table identifier or tabid.
- Showed how to join a table to itself with a self-join to compare values in a column with other values in the same column and to identify duplicates.
- Introduced the keyword OUTER, explained how an outer join treats two or more tables asymmetrically, and provided examples of the four kinds of outer join.

- Described how to create correlated and uncorrelated subqueries by nesting a `SELECT` statement in the `WHERE` clause of another `SELECT` statement and showed the use of aggregate functions in subqueries.
- Demonstrated the use of the keywords `ALL`, `ANY`, `EXISTS`, `IN`, and `SOME` in creating subqueries, and the effect of adding the keyword `NOT` or a relational operator.
- Discussed the set operations union, intersection, and difference.
- Showed how to use the `UNION` and `UNION ALL` keywords to create compound queries consisting of two or more `SELECT` statements.

Summary

Modifying Data

Chapter Overview	3
Statements That Modify Data	3
Deleting Rows	4
Deleting All Rows of a Table	4
Deleting a Known Number of Rows	4
Deleting an Unknown Number of Rows	5
Complicated Delete Conditions	6
Inserting Rows	6
Single Rows	7
Multiple Rows and Expressions	9
Restrictions on the Insert-Selection	10
Updating Rows	11
Selecting Rows to Update	12
Updating with Uniform Values	13
Impossible Updates	13
Updating with Selected Values	14
Database Privileges	15
Displaying Table Privileges	16
Data Integrity	17
Entity Integrity	18
Semantic Integrity	18
Referential Integrity	19
Using the ON DELETE CASCADE Option	21
Interrupted Modifications	22
The Transaction	23
Transaction Logging	24
Logging and Cascading Deletes	24
Specifying Transactions	24

Archives and Logs	25
Archiving with INFORMIX-SE	26
Archiving with INFORMIX-OnLine Dynamic Server	26
Concurrency and Locks	27
Data Replication	28
INFORMIX-OnLine Dynamic Server Data Replication	29
Summary	29

Chapter Overview

Modifying data is fundamentally different from querying data. Querying data involves examining the contents of tables. Modifying data involves *changing* the contents of tables.

Think about what happens if the system hardware or software fails during a query. In this case, the effect on the application can be severe but the database itself is unharmed. However, if the system fails while a modification is under way, the state of the database is in doubt. Obviously, this can have far-reaching implications. Before you delete, insert, or update rows in a database, ask yourself the following questions:

- Is user access to the database and its tables secure, that is, are specific users given limited database and table-level privileges?
- Does the modified data preserve the existing integrity of the database?
- Are systems in place that make the database relatively immune to external events that might cause system or hardware failures?

If you are unable to answer yes to each of these questions, do not panic. Solutions to all these problems are built in to the Informix database servers. After an introduction to the statements that modify data, this chapter discusses these solutions. Chapters 8 through 11 talk about these topics in greater detail.

Statements That Modify Data

The following three statements modify data:

- DELETE
- INSERT
- UPDATE

Although these SQL statements are relatively simple when compared with the more advanced SELECT statements, use them carefully because they change the contents of the database.

Deleting Rows

The DELETE statement removes any row or combination of rows from a table. You cannot recover a deleted row after the transaction is committed. (Transactions are discussed under “Interrupted Modifications” on page 4-22. For now, think of a transaction and a statement as the same thing.)

When deleting a row, you must also be careful to delete any rows of other tables whose values depend on the deleted row. If, however, your database enforces referential constraints, you can use the ON DELETE CASCADE option of the CREATE TABLE or ALTER TABLE statements to allow deletes to cascade from one table in a relationship to another. For more information on referential constraints and the ON DELETE CASCADE option, refer to the section “Referential Integrity” on page 4-19.

Deleting All Rows of a Table

The DELETE statement specifies a table and usually contains a WHERE clause that designates the row or rows that are to be removed from the table. If the WHERE clause is left out, all rows are deleted. *Do not execute the following statement:*

```
DELETE FROM customer
```

Because this DELETE statement does not contain a WHERE clause, all rows from the **customer** table are deleted. If you attempt an unconditional delete using the **DB-Access** or **INFORMIX-SQL** menu options, the program warns you and asks for confirmation. However, an unconditional delete from within a program can occur without warning.

Deleting a Known Number of Rows

The WHERE clause in a DELETE statement has the same form as the WHERE clause in a SELECT statement. You can use it to designate exactly which row or rows should be deleted. You can delete a customer with a specific customer number, as shown in the following example:

```
DELETE FROM customer WHERE customer_num = 175
```

In this example, because the **customer_num** column has a unique constraint, you are sure that no more than one row is deleted.

Deleting an Unknown Number of Rows

You can also choose rows based on nonindexed columns, as shown in the following example:

```
DELETE FROM customer WHERE company = 'Druid Cyclery'
```

Because the column tested does not have a unique constraint, this statement might delete more than one row. (Druid Cyclery may have two stores, both with the same name but different customer numbers.)

You can find out how many rows might be affected by a DELETE statement by selecting the count of qualifying rows from the **customer** table for Druid Cyclery.

```
SELECT COUNT(*) FROM customer WHERE company = 'Druid Cyclery'
```

You can also select the rows and display them, to be sure they are the ones you want to delete.

Using a SELECT statement as a test is only an approximation, however, when the database is available to multiple users concurrently. Between the time you execute the SELECT statement and the subsequent DELETE statement, other users could have modified the table and changed the result. In this example, another user might perform the following actions:

- Insert a new row for another customer named Druid Cyclery
- Delete one or more of the Druid Cyclery rows before you do so
- Update a Druid Cyclery row to have a new company name, or update some other customer to have the name Druid Cyclery

Although it is not *likely* that other users would do these things in that brief interval, the possibility does exist. This same problem affects the UPDATE statement. Ways of addressing this problem are discussed under “Concurrency and Locks” on page 4-27, and in greater detail in Chapter 7, “Programming for a Multiuser Environment.”

Another problem you may encounter is a hardware or software failure before the statement finishes. In this case, the database may have deleted no rows, some rows, or all specified rows. The *state* of the database is unknown, which is undesirable. You can prevent this situation by using transaction logging, as discussed in “Interrupted Modifications” on page 4-22.

Complicated Delete Conditions

The WHERE clause in a DELETE statement can be almost as complicated as the one in a SELECT statement. It can contain multiple conditions connected by AND and OR, and it may contain subqueries.

Suppose you discover that some rows of the **stock** table contain incorrect manufacturer codes. Rather than update them, you want to delete them so they can be reentered. You know that these rows, unlike the correct ones, have no matching rows in the **manufact** table. This allows you to write a DELETE statement such as the one shown in the following example:

```
DELETE FROM stock
  WHERE 0 = (SELECT COUNT(*) FROM manufact
             WHERE manufact.manu_code = stock.manu_code)
```

The subquery counts the number of rows of **manufact** that match; the count is 1 for a correct row of **stock** and 0 for an incorrect one. The latter rows are chosen for deletion.

One way to develop a DELETE statement with a complicated condition is to first develop a SELECT statement that returns precisely the rows to be deleted. Write it as SELECT *; when it returns the desired set of rows, change SELECT * to read DELETE and execute it once more.

The WHERE clause of a DELETE statement cannot use a subquery that tests the same table. That is, when you delete from **stock**, you cannot use a subquery in the WHERE clause that also selects from **stock**.

The key to this rule is in the FROM clause. If a table is named in the FROM clause of a DELETE statement, it cannot also appear in the FROM clause of a subquery of the DELETE statement.

Inserting Rows

The INSERT statement adds a new row, or rows, to a table. The statement has two basic functions: It can create a single new row using column values you supply, or it can create a group of new rows using data selected from other tables.

Single Rows

In its simplest form, the INSERT statement creates one new row from a list of column values, and puts that row in the table. The following statement is an example of adding a row to the **stock** table:

```
INSERT INTO stock
VALUES(115, 'PRC', 'tire pump', 108, 'box', '6/box')
```

The **stock** table has the following columns:

stock_num (a number identifying the type of merchandise)

manu_code (a foreign key to the **manufact** table)

description (a description of the merchandise)

unit_price (the unit price of the merchandise)

unit (of measure)

unit_descr (characterizing the unit of measure)

Notice that the values listed in the VALUES clause in the preceding example have a one-to-one correspondence with the columns of this table. To write a VALUES clause, you must know the columns of the tables as well as their sequence from first to last.

Possible Column Values

The VALUES clause accepts *only* constant values, *not* expressions. You can supply the following values:

- Literal numbers
- Literal datetime values
- Literal interval values
- Quoted strings of characters
- The word NULL for a null value
- The word TODAY for today's date
- The word CURRENT for the current date and time
- The word USER for your user name
- The word DBSERVERNAME (or SITENAME) for the name of the computer where the database server is running

Some columns of a table might not allow null values. If you attempt to insert NULL in such a column, the statement is rejected. Or a column in the table may not permit duplicate values. If you specify a value that is a duplicate of

one already in such a column, the statement is rejected. Some columns may even *restrict* the possible column values allowed. These restrictions are placed on columns using data integrity constraints. For more information on data restrictions, see the section “Database Privileges” on page 4-15.

Only one column in a table can have the SERIAL data type. The database server generates values for a serial column. To make this happen, when you insert values, specify the value zero for the serial column. The database server generates the next actual value in sequence. Serial columns do not allow null values.

You can specify a nonzero value for a serial column (as long as it does not duplicate any existing value in that column) and the database server uses the value. However, that nonzero value may set a new starting point for values that the database server generates. The next value the database server generates for you is one greater than the maximum value in the column.

Do not specify the currency symbols for columns that contain money values. Just specify the numeric value of the amount.

The database server can convert between numeric and character data types. You can give a string of numeric characters (for example, '-0075.6') as the value of a numeric column. The database server converts the numeric string to a number. An error occurs only if the string does not represent a number.

You can specify a number or a date as the value for a character column. The database server converts that value to a character string. For example, if you specify TODAY as the value for a character column, a character string representing today's date is used. (The format used is specified by the DBDATE environment variable.)

Listing Specific Column Names

You do not have to specify values for every column. Instead, you can list the column names after the table name and then supply values for only those columns you named. The following example shows a statement that inserts a new row into the **stock** table:

```
INSERT INTO stock (stock_num, description, unit_price, manu_code)
VALUES (115, 'tyre pump', 114, 'SHM')
```

Notice that only the data for the stock number, description, unit price, and manufacturer code is provided. The database server supplies the following values for the remaining columns:

- It generates a serial number for an unlisted serial column.
- It generates a default value for a column with a specific default associated with it.
- It generates a null value for any column that allows nulls but does not specify a default value for any column that specifies null as the default value.

This means that you must list and supply values for all columns that do not specify a default value or do not permit nulls. However, you can list the columns in any order—as long as the values for those columns are listed in the same order. For information about setting a default value for a column, see Chapter 9, “Implementing Your Data Model.”

After the INSERT statement is executed, the following new row is inserted into the **stock** table:

stock_num	manu_code	description	unit_price	unit	unit_descr
115	SHM	tyre pump	114		

Both the **unit** and **unit_descr** are blank, indicating that null values are in those two columns. Because the **unit** column permits nulls, one can only guess the number of tire pumps that were purchased for \$114. Of course, if a default value of “box” was specified for this column, then “box” would be the unit of measure. In any case, when inserting values into specific columns of a table, pay attention to what data is needed for that row.

Multiple Rows and Expressions

The other major form of the INSERT statement replaces the VALUES clause with a SELECT statement. This feature allows you to insert the following data:

- Multiple rows with only one statement (a row is inserted for each row returned by the SELECT statement)
- Calculated values (the VALUES clause only permits constants) because the select list can contain expressions

For example, suppose a follow-up call is required for every order that has been paid for but not shipped. The INSERT statement in the following example finds those orders and inserts a row in **cust_calls** for each order:

```
INSERT INTO cust_calls (customer_num, call_descr)
  SELECT customer_num, order_num FROM orders
     WHERE paid_date IS NOT NULL
     AND ship_date IS NULL
```

This SELECT statement returns two columns. The data from these columns (in each selected row) is inserted into the named columns of the **cust_calls** table. Then, an order number (from **order_num**, a serial column) is inserted into the call description, which is a character column. Remember that the database server allows you to insert integer values into a character column. It automatically converts the serial number to a character string of decimal digits.

Restrictions on the Insert-Selection

The following list contains the restrictions on the SELECT statement for inserting rows:

- It cannot contain an INTO clause.
- It cannot contain an INTO TEMP clause.
- It cannot contain an ORDER BY clause.
- It cannot refer to the table into which you are inserting rows.

The INTO, INTO TEMP, and ORDER BY clause restrictions are minor. The INTO clause is not useful in this context. (It is discussed in Chapter 5, “Programming with SQL.”) You can work around the INTO TEMP clause restriction by first selecting the data you want to insert into a temporary table and then inserting the data from the temporary table with the INSERT statement. Likewise, the lack of an ORDER BY clause is not important. If you need to ensure that the new rows are physically ordered in the table, you can first select them into a temporary table and order it, and then insert from the temporary table. You can also apply a physical order to the table using a clustered index after all insertions are done.

The last restriction is more serious because it prevents you from naming the *same* table in both the INTO clause of the INSERT statement and the FROM clause of the SELECT statement. (This saves the database server from getting into an endless loop in which each inserted row is reselected and reinserted.) In some cases, however, you may want to do this. For example, suppose that you have learned that the Nikolus company supplies the same products as

the Anza company, but at half the price. You want to add rows to the **stock** table to reflect this. Optimally, you want to select data from all the Anza stock rows and reinsert it with the Nikolus manufacturer code. However, you cannot select from the same table into which you are inserting.

You can get around this restriction by selecting the data you want to insert into a temporary table. Then, select from that table in the INSERT statement. The following statements are required to accomplish this:

```
SELECT stock_num, 'HSK' temp_manu, description, unit_price/2
       half_price, unit, unit_descr FROM stock
WHERE manu_code = 'ANZ'
      AND stock_num < 110
      INTO TEMP anzrows;

INSERT INTO stock SELECT * FROM anzrows;

DROP TABLE anzrows;
```

This SELECT statement takes existing rows from **stock** and substitutes a literal value for the manufacturer code and a computed value for the unit price. These rows are then saved in a temporary table, **anzrows**, which is immediately inserted into the **stock** table.

When you insert multiple rows, a risk exists that one of the rows contains invalid data that might cause the database server to report an error. When this happens, the statement terminates early. Even if no error occurs, a very small risk exists that there might be a hardware or software failure while the statement is executing (for example, the disk might fill up).

In either event, you cannot easily tell how many new rows were inserted. If you repeat the statement in its entirety, you might create duplicate rows—or you might not. Because the database is in an unknown state, you cannot know what to do. The answer lies in using transactions, as discussed in “Interrupted Modifications” on page 4-22.

Updating Rows

You use the UPDATE statement to change the contents of one or more columns in one or more existing rows of a table. This statement takes two fundamentally different forms. One lets you assign specific values to columns by name; the other lets you assign a list of values (that might be returned by a SELECT statement) to a list of columns. In either case, if you are

updating rows and some of the columns have data integrity constraints, the data you change must be within the constraints placed on those columns. For more information, refer to “Database Privileges” on page 4-15.

Selecting Rows to Update

Either form of the UPDATE statement can end with a WHERE clause that determines which rows are modified. If you omit it, all rows are modified. The WHERE clause can be quite complicated to select the precise set of rows that need changing. The only restriction on it is that the table that you are updating cannot be named in the FROM clause of a subquery.

The first form of an UPDATE statement uses a series of assignment clauses to specify new column values, as shown in the following example:

```
UPDATE customer
  SET fname = 'Barnaby', lname = 'Dorfler'
 WHERE customer_num = 103
```

The WHERE clause selects the row to be updated. In the **stores6** database, the **customer.customer_num** column is the primary key for that table, so this statement can update no more than one row.

You can also use subqueries in the WHERE clause. Suppose that the Anza Corporation issues a safety recall of their tennis balls. As a result, any unshipped orders that include stock number 6 from manufacturer ANZ must be put on back order, as shown in the following example:

```
UPDATE orders
  SET backlog = 'y'
 WHERE ship_date IS NULL
 AND order_num IN
   (SELECT DISTINCT items.order_num FROM items
    WHERE items.stock_num = 6
     AND items.manu_code = 'ANZ')
```

This subquery returns a column of order numbers (zero or more). The UPDATE operation then tests each row of **orders** against the list and performs the update if that row matches.

Updating with Uniform Values

Each assignment after the keyword **SET** specifies a new value for a column. That value is applied uniformly to every row that you update. In the examples in the previous section, the new values were constants, but you can assign any expression, including one based on the column value itself. Suppose the manufacturer code **HRO** has raised all prices by 5 percent, and you must update the **stock** table to reflect this. You would use a statement such as the one shown in the following example:

```
UPDATE stock
  SET unit_price = unit_price * 1.05
  WHERE manu_code = 'HRO'
```

You can also use a subquery as part of the assigned value. When a subquery is used as an element of an expression, it must return exactly one value (one column and one row). Suppose that you decide that for any stock number, you must charge a higher price than any manufacturer of that product. You need to update the prices of all unshipped orders. The **SELECT** statements in the following example specify the criteria:

```
UPDATE items
  SET total_price = quantity *
    (SELECT MAX (unit_price) FROM stock
     WHERE stock.stock_num = items.stock_num)
  WHERE items.order_num IN
    (SELECT order_num FROM orders
     WHERE ship_date IS NULL)
```

The first **SELECT** statement returns a single value—the highest price in the **stock** table for a particular product. This is a correlated subquery; because when a value from **items** appears in its **WHERE** clause, you must execute it for every row that you update.

The second **SELECT** statement produces a list of the order numbers of unshipped orders. It is an uncorrelated subquery that is executed once.

Impossible Updates

Restrictions exist on the use of subqueries when you modify data. In particular, you cannot query the table that is being modified. You *can* refer to the present value of a column in an expression, as in the example in which the **unit_price** column was incremented by 5 percent. You *can* refer to a value of

a column in a WHERE clause in a subquery, as in the example that updated the **stock** table, in which the **items** table is updated and **items.stock_num** is used in a join expression.

The need to update and query a table at the same time does not occur often in a well-designed database. (Database design is covered in Chapters 8 through 11 of this manual.) However, you may want to update and query at the same time when a database is first being developed, before its design has been carefully thought through. A typical problem arises when a table inadvertently and incorrectly contains a few rows with duplicate values in a column that should be unique. You might want to delete the duplicate rows or update only the duplicate rows. Either way, a test for duplicate rows inevitably requires a subquery, which is not allowed in an UPDATE statement or DELETE statement. Chapter 6, “Modifying Data Through SQL Programs,” discusses how to use an *update cursor* to perform this kind of modification.

Updating with Selected Values

The second form of UPDATE statement replaces the list of assignments with a single bulk assignment, in which a list of columns is set equal to a list of values. When the values are simple constants, this form is nothing more than the form of the previous example with its parts rearranged, as shown in the following example:

```
UPDATE customer
  SET (fname, lname) = ('Barnaby', 'Dorfler')
  WHERE customer_num = 103
```

No advantage exists to writing the statement this way. In fact, it is harder to read because it is not obvious which values are assigned to which columns.

However, when the values to be assigned come from a single SELECT statement, this form makes sense. Suppose that changes of address are to be applied to several customers. Instead of updating the **customer** table each time a change is reported, the new addresses are collected in a single temporary table named **newaddr**. It contains columns for the customer number and the address-related fields of the **customer** table. Now the time comes to apply all the new addresses at once. (This idea of collecting routine changes in a

separate table and applying them in a batch, outside of peak hours, is one of the performance techniques covered in Chapter 10, “Tuning Your Data Model.”)

```
UPDATE customer
  SET (address1, address2, city, state, zipcode) =
      ((SELECT address1, address2, city, state, zipcode
        FROM newaddr
        WHERE newaddr.customer_num=customer.customer_num))
  WHERE customer_num IN
      (SELECT customer_num FROM newaddr)
```

Notice that the values for multiple columns are produced by a single SELECT statement. If you rewrite this example in the other form, with an assignment for each updated column, you must write five SELECT statements, one for each column to be updated. Not only is such a statement harder to write, it also takes much longer to execute.

Note: In INFORMIX-4GL and the SQL API programs, you can use record or host variables to update values. For more information, refer to Chapter 5, “Programming with SQL.”

Database Privileges

Two levels of privileges exist in a database: database-level privileges and table-level privileges. When you create a database, you are the only one who can access it until you, as the owner (or Database Administrator [DBA]) of the database, grant database-level privileges to others. When you create a table in a database that is not ANSI-compliant, all users have access privileges to the table until you, as the owner of the table, revoke table-level privileges from specific users.

The following list contains database-level privileges:

Connect privilege	allows you to open a database, issue queries, and create and place indexes on temporary tables.
Resource privilege	allows you to create permanent tables.
DBA privilege	allows you to perform several additional functions as the DBA.

Seven table-level privileges exist. However, only the first four are covered here:

Select privilege	is granted on a table-by-table basis and allows you to select rows from a table (This privilege can be limited by specific columns in a table.)
Delete privilege	allows you to delete rows.
Insert privilege	allows you to insert rows.
Update privilege	allows you to update existing rows (that is, to change their content).

The people who create databases and tables often grant the Connect and Select privileges to *public* so that all users have them. If you can query a table, you have at least the Connect and Select privileges for that database and table. For more information about *public*, see “The Users and the Public” on page 11-6.

You need the other table-level privileges to modify data. The owners of tables often withhold these privileges or grant them only to specific users. As a result, you may not be able to modify some tables that you can query freely.

Because these privileges are granted on a table-by-table basis, you can have only Insert privileges on one table and only Update privileges on another, for example. The Update privileges can be restricted even further to specific columns *in* a table.

Chapter 11, “Granting and Limiting Access to Your Database,” discusses granting privileges from the standpoint of the DBA. A complete list of privileges and a summary of the GRANT and REVOKE statements can be found in Chapter 1 of the *Informix Guide to SQL: Syntax*.

Displaying Table Privileges

If you are the owner of a table (that is, if you created it), you have all privileges on that table. Otherwise, you can determine the privileges you have for a certain table by querying the system catalog. The system catalog consists of system tables that describe the database structure. The privileges granted on each table are recorded in the **systabauth** system table. To display these privileges, you must also know the unique identifier number of the table. This number is specified in the **systables** system table. So, to display privileges granted on the **orders** table, you might enter the following SELECT statement:

```
SELECT * FROM systabauth
      WHERE tabid = (SELECT tabid FROM systables
                    WHERE tablename = 'orders')
```

The output of the query resembles the following display:

grantor	grantee	tabid	tabauth
tfecit	mutator	101	su-i-x--
tfecit	procrustes	101	s--idx--
tfecit	public	101	s--i-x--

The grantor is the user who *grants* the privilege. The grantor is usually the owner of the table but can be another user empowered by the grantor. The grantee is the user to whom the privilege is granted, and the grantee *public* means “any user with Connect privilege.” If your user name does not appear, you have only those privileges granted to public.

The **tabauth** column specifies the privileges granted. The letters in each row of this column are the initial letters of the privilege names except that *i* means Insert and *x* means Index. In this example, public has Select, Insert, and Index privileges. Only the user *mutator* has Update privileges, and only the user *procrustes* has Delete privileges.

Before the database server performs any action for you (for example, execute a DELETE statement), it performs a query similar to the preceding one. If you are not the owner of the table, and if it cannot find the necessary privilege on the table for your user name or for public, it refuses to perform the operation.

Data Integrity

The INSERT, UPDATE, and DELETE statements modify data in an existing database. Whenever you modify existing data, the *integrity* of the data can be affected. For example, an order for a nonexistent product could be entered into the **orders** table. Or a customer with outstanding orders could be deleted from the **customer** table. Or the order number could be updated in the **orders** table and *not* in the **items** table. In each of these cases, the integrity of the stored data is lost.

Data integrity is actually made up of three parts:

- Entity integrity
Each row of a table has a unique identifier.
- Semantic integrity
The data in the columns properly reflects the types of information the column was designed to hold.

- Referential integrity

The relationships between tables are enforced.

Well-designed databases incorporate these principles so that when you modify data, the database itself prevents you from doing anything that might harm the data integrity.

Entity Integrity

An entity is any person, place, or thing to be recorded in a database. Each entity represents a table, and each row of a table represents an instance of that entity. For example, if *order* is an entity, the **orders** table represents the idea of order and *each row* in the table represents a specific order.

To identify each row in a table, the table must have a primary key. The primary key is a unique value that identifies each row. This requirement is called the *entity integrity constraint*.

For example, the **orders** table primary key is **order_num**. The **order_num** column holds a unique system-generated order number for each row in the table. To access a row of data in the **orders** table you can use the following SELECT statement:

```
SELECT * FROM orders WHERE order_num = 1001
```

Using the order number in the WHERE clause of this statement enables you to access a row easily because the order number uniquely identifies that row. If the table allowed duplicate order numbers, it would be almost impossible to access one single row, because all other columns of this table allow duplicate values.

Refer to Chapter 8, “Building Your Data Model,” for more information on primary keys and entity integrity.

Semantic Integrity

Semantic integrity ensures that data entered into a row reflects an allowable value for that row. This means that the value must be within the *domain*, or allowable set of values, for that column. For example, the **quantity** column of the **items** table permits only numbers. If a value outside the domain can be entered into a column, the semantic integrity of the data is violated.

Semantic integrity is enforced using the following constraints:

- Data type

The data type defines the types of values that you can store in a column. For example, the data type `SMALLINT` allows you to enter values from -32,767 to 32,767 into a column.

- Default value

The default value is the value inserted into the column when an explicit value is not specified. For example, the `user_id` column of the `cust_calls` table defaults to the login name of the user if no name is entered.

- Check constraint

The check constraint specifies conditions on data inserted into a column. Each row inserted into a table must meet these conditions. For example, the `quantity` column of the `items` table may check for quantities greater than or equal to 1.

For more information on using semantic integrity constraints in database design, refer to “Defining the Domains” on page 9-3.

Referential Integrity

Referential integrity refers to the relationship *between* tables. Because each table in a database must have a primary key, it is possible that this primary key appears in other tables because of its relationship to data within those tables. When a primary key from one table appears in another table, it is called a foreign key.

Foreign keys *join* tables and establish dependencies between tables. Tables can form a hierarchy of dependencies in such a way that if you change or delete a row in one table, you destroy the meaning of rows in other tables. For example, in Figure 4-1, the `customer_num` column of the `customer` table is a primary key for that table and a foreign key in the `orders` and `cust_call` tables. Customer number 106, George Watson, is *referenced* in both the `orders` and `cust_calls` tables. If customer 106 is deleted from the `customer` table, the link between the three tables and this particular customer is destroyed.

When you delete a row containing a primary key or update it with a different primary key, you destroy the meaning of any rows that contain that value as a foreign key. Referential integrity is the logical dependency of a foreign key on a primary key. The *integrity* of a row that contains a foreign key depends on the integrity of the row that it *references*—the row that contains the matching primary key.

By default, **INFORMIX-OnLine Dynamic Server** does not allow you to violate referential integrity and gives you an error message if you attempt to delete rows from the parent table before you delete rows from the child table. You can, however, use the **ON DELETE CASCADE** option to cause deletes from a parent table to trip deletes on child tables. See “Using the **ON DELETE CASCADE** Option” on page 4-21.

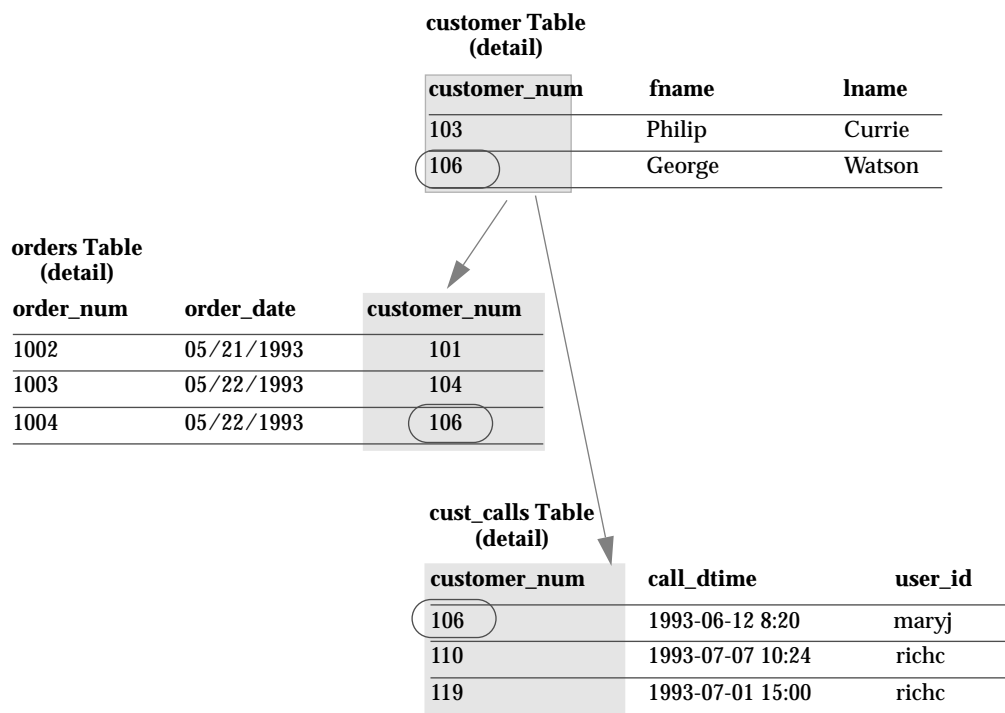


Figure 4-1 Referential integrity in the stores6 database

You can define primary and foreign keys, and the relationship between them, using the **CREATE TABLE** and **ALTER TABLE** statements. For more information on these statements, refer to Chapter 1 of the *Informix Guide to SQL: Syntax*. For information on building data models using primary and foreign keys, refer to Chapter 8, “Building Your Data Model.”

Using the ON DELETE CASCADE Option

You can maintain referential integrity when deleting rows from a primary key for a table by using the ON DELETE CASCADE option in the REFERENCES clause of the CREATE TABLE and ALTER TABLE statements. This option allows you to delete a row from a parent table and its corresponding rows in matching child tables with a single delete command.

Locking During Cascading Deletes

During deletes, locks are held on all qualifying rows of the parent and child tables. When you specify a delete, the delete requested from the parent table occurs before any referential actions are performed.

What Happens to Multiple Children Tables

If you have a parent table with two child constraints, one child with cascading deletes specified and one child without cascading deletes, and you attempt to delete a row from the parent table that applies to both child tables, then the DELETE statement fails and no rows are deleted from either the parent or child tables.

Logging Must Be Turned On

You must turn logging on in your current database for cascading deletes to work. Logging and cascading deletes are discussed in “Transaction Logging” on page 4-24.

Example

Say you have two tables with referential integrity rules applied, a parent table, **accounts**, and a child table, **sub_accounts**. The following CREATE TABLE statements define the referential constraints:

```
CREATE TABLE accounts (  
  acc_num SERIAL primary key,  
  acc_type INT,  
  acc_descr CHAR(20));  
  
CREATE TABLE sub_accounts (  
  sub_acc INTEGER primary key,  
  ref_num INTEGER REFERENCES references accounts (acc_num) ON DELETE CASCADE,  
  sub_descr CHAR(20));
```

Notice that the primary key of the **accounts** table, the **acc_num** column, uses a SERIAL data type, and the foreign key of the **sub_accounts** table, the **ref_num** column, uses an INTEGER data type. Combining the SERIAL data type on the primary key and the INTEGER data type on the foreign key is allowed. This is the only condition where you can mix and match data types. The SERIAL data type is an INTEGER and the values for the column are automatically generated by the database. All other primary and foreign key combinations must match explicitly. For example, a primary key defined as CHAR must match a foreign key defined as CHAR.

To delete a row from the **accounts** table that will cascade a delete to the **sub_accounts** table, you must turn on logging. After logging is turned on, you can delete the account number 2 from both tables, as shown in the following example:

```
DELETE FROM accounts WHERE acc_num = 2
```

Restrictions on Cascading Deletes

You can use cascading deletes for most deletes, including deletes on self-referencing and cyclic queries. The only exception is correlated subqueries. In correlated subqueries, the subquery (or inner SELECT) is correlated when the value it produces depends on a value produced by the outer SELECT statement that contains it. If you have implemented cascading deletes, you cannot write deletes that use a child table in the correlated subquery. You receive an error when you attempt to delete from a correlated subquery.

Interrupted Modifications

Even if all the software is error-free and all the hardware is utterly reliable, the world outside the computer can interfere. It is possible for lightning to strike the building, interrupting the electrical supply and stopping the computer in the middle of your UPDATE statement. It is more likely, however, that a disk fills up or a user supplies incorrect data, causing your multirow insert to stop early with an error. In any case, as you are modifying data, you must assume that some unforeseen event can interrupt the modification.

When a modification is interrupted by an external cause, you cannot be sure how much of the operation was completed. Even in a single-row operation, you cannot know whether the data reached the disk or the indexes were properly updated.

If multirow modifications are a problem, multistatement ones are worse. It does not help that they are usually embedded in programs so that you do not see the individual SQL statements being executed. For example, the job of entering a new order in the **stores6** database requires you to perform the following steps:

- Insert a row in the **orders** table. (This generates an order number.)
- For each item ordered, insert a row in the **items** table.

Two ways to program an order-entry application exist. One way is to make it completely interactive so that the program inserts the first row immediately, and then inserts each item as the operator enters data. This is the wrong approach because it exposes the operation to the possibility of many more unforeseen events: the customer's telephone disconnecting, the operator pressing the wrong key, the operator's terminal losing power, and so on.

The right way to build an order-entry application is described in the following list:

- Accept all the data interactively.
- Validate the data and expand it (by looking up codes in **stock** and **manufact**, for example).
- Display the information on the screen for inspection.
- Wait for the operator to make a final commitment.
- Perform the insertions quickly.

Even when this is done, sometimes an unforeseen circumstance can halt the program after it inserts the order but before it finishes inserting the items. If that happens, the database is in an unpredictable condition: Its *data integrity* is compromised.

The Transaction

The solution to all these potential problems is called the *transaction*. A transaction is a sequence of modifications that either must be accomplished completely or not at all. The database server guarantees that operations performed within the bounds of a transaction are either completely and perfectly committed to disk, or the database is restored to the state it was in before the transaction started.

The transaction is not merely protection against unforeseen failures; it also offers a program a way to escape when the program detects a logical error. (This is discussed further in Chapter 6, "Modifying Data Through SQL Programs.")

Transaction Logging

The database server can keep a record of each change that it makes to the database during a transaction. If something happens to cancel the transaction, the database server automatically uses the records to reverse the changes. Many things can make a transaction fail. The program that issues the SQL statements can crash or be terminated, or there might be a hardware or software failure in any other component of the system. As soon as the database server discovers that the transaction failed, which might be only after the computer and the database server are restarted, it returns the database to the state it was in before the transaction began using the records from the transaction.

The process of keeping records of transactions is called *transaction logging*, or simply *logging*. The records of the transactions, called *log records*, are stored in a portion of disk space separate from the database. In **INFORMIX-OnLine Dynamic Server**, this space is called the *logical log* (because the log records represent logical units of the transactions). In **INFORMIX-SE**, the space used to store log records is called the transaction log file.

Databases do not generate transaction records automatically. The database administrator (or the **INFORMIX-OnLine Dynamic Server** administrator, for **OnLine** databases) decides whether to make a database use transaction logging. Without transaction logging, you cannot roll back transactions.

Logging and Cascading Deletes

Logging must be turned on in your database for cascading deletes to work. This is because when you specify a cascading delete, the delete is first performed on the primary key of the parent table. If the system crashes after the rows of the primary key of the parent table are performed but before the rows of the foreign key of the child table are deleted, referential integrity is violated. If logging is turned off, even temporarily, deletes do not cascade. After logging is turned back on, however, deletes can cascade again. Turn logging on with the CREATE DATABASE statement for **INFORMIX-OnLine Dynamic Server** database servers.

Specifying Transactions

The boundaries of transactions are specified with SQL statements. You can do this in two ways. In the most common style, you specify the start of a multi-statement transaction by executing the BEGIN WORK statement. In databases

that are created with the `MODE ANSI` option, no need exists to mark the beginning of a transaction. One is always in effect; you indicate only the end of each transaction.

In both styles, you specify the end of a successful transaction by executing the `COMMIT WORK` statement. This statement tells the database server that you reached the end of a series of statements that must succeed together. The database server does whatever is necessary to make sure that all modifications are properly completed and committed to disk.

It is also possible for a program to cancel a transaction deliberately. It does so by executing the `ROLLBACK WORK` statement. This statement asks the database server to cancel the current transaction and undo any changes.

For example, an order-entry application can use a transaction when creating a new order in the following ways:

- Accept all data interactively
- Validate and expand it
- Wait for the operator to make a final commitment
- Execute `BEGIN WORK`
- Insert rows in the **orders** and **items** tables, checking the error code returned by the database server
- If there were no errors, execute `COMMIT WORK`, otherwise execute `ROLLBACK WORK`

If any external failure prevents the transaction from being completed, the partial transaction rolls back when the system restarts. In all cases, the database is in a predictable state: Either the new order is completely entered, or it is not entered at all.

Archives and Logs

By using transactions, you can ensure that the database is always in a consistent state and that your modifications are properly recorded on disk. But the disk itself is not perfectly safe. It is vulnerable to mechanical failures and to flood, fire, and earthquake. The only safeguard is to keep multiple copies of the data. These redundant copies are called *archive* copies.

The transaction log (also called the logical log) complements the archive copy of a database. Its contents are a history of all modifications that occurred since the last time the database was archived. If it should ever be necessary to restore the database from the archive copy, the transaction log can be used to roll the database forward to its most recent state.

Archiving with INFORMIX-SE

If a database is stored in operating system files (**INFORMIX-SE**), archive copies are made using the normal methods for making backup copies in your operating system. Only two special considerations exist for databases.

The first is a practical consideration: A database can grow to great size. It may become the largest file or set of files in the system. It can also be awkward or very time consuming to make a copy of it. You may need a special procedure for copying the database, separate from the usual backup procedures, and the job may not be done too frequently.

The second consideration is the special relationship between the database and the transaction log file. An archive copy is an image of the database at one instant. The log file contains the history of modifications that were made during that instant. It is important that those two instants are identical; in other words, it is important to start a new transaction log file immediately upon making an archive copy of the database. Then, if you must restore the database from the archive tape, the transaction log contains exactly the history needed to bring it forward in time from that instant to the latest update.

The statement that applies a log to a restored database is **ROLLFORWARD DATABASE**. You start a new log file by using whatever operating system commands are needed to delete the file and re-create it empty or simply to set the length of the file to zero.

A transaction log file can grow to extreme size. If you update a row ten times, just one row exists in the database—but ten update events are recorded in the log file. If the size of the log file is a problem, you can start a fresh log. Choose a time when the database is not being updated (so no transactions are active), and copy the existing log to another medium. That copy represents all modifications for some period of time; preserve it carefully. Then start a new log file. If you ever have to restore the database, you must apply all the log files in their correct sequence.

Archiving with INFORMIX-OnLine Dynamic Server

The **OnLine** database server contains elaborate features to support archiving and logging. They are described in the *INFORMIX-OnLine Dynamic Server Archive and Backup Guide*.

Conceptually, the facilities of **OnLine** are similar to those already described for **INFORMIX-SE**, but they are more elaborate for the following reasons:

- **OnLine** has very stringent requirements for performance and reliability (for example, it supports making archive copies while databases are in use).
- **OnLine** manages its own disk space that is devoted to logging.
- It performs logging concurrently for all databases using a limited set of log files. The log files can be copied to another medium (backed up) while transactions are active.

These facilities are usually managed from a central location by the **OnLine** administrator, so the users of the databases never have to be concerned with them.

If you want to make a personal archive copy of a single database or table that is held by **OnLine**, you can do it with the **onunload** utility. This program copies a table or a database to tape. Its output consists of binary images of the disk pages as they were stored in **OnLine**. As a result, the copy can be made very quickly, and the corresponding **onload** program can restore the file very quickly. However, the data format is not meaningful to any other programs.

If your **OnLine** administrator is using ON-Archive to create archives and back up logical logs, you also may be able to create your own backup copies using ON-Archive. See your *INFORMIX-OnLine Dynamic Server Archive and Backup Guide* for more information.

Concurrency and Locks

If your database is contained in a single-user workstation, without a network connecting it to other machines, concurrency is unimportant. In all other cases, you must allow for the possibility that, while your program is modifying data, another program is also reading or modifying the same data. This is *concurrency*: two or more independent uses of the same data at the same time.

A high level of concurrency is crucial to good performance in a multiuser database system. Unless controls exist on the use of data, however, concurrency can lead to a variety of negative effects. Programs could read obsolete data; modifications could be lost even though it seems they were entered successfully.

The database server prevents errors of this kind by imposing a system of *locks*. A lock is a claim, or reservation, that a program can place on a piece of data. The database server guarantees that, as long as the data is locked, no other program can modify it. When another program requests the data, the database server either makes the program wait or turns it back with an error.

You use two statements to control the effect that locks have on your data access: `SET LOCK MODE` and `SET ISOLATION`. You can understand the details of these statements are best understood after a discussion on the use of *cur*-*sors* from within programs. Cursors are covered in Chapter 5, “Programming with SQL,” and Chapter 6, “Modifying Data Through SQL Programs.” Also see Chapter 7, “Programming for a Multiuser Environment,” for more information about locking.

Data Replication

Data replication, in the broadest sense of the term, is when database objects have more than one representation at more than one distinct site. For example, a database that is copied to a database server on a different machine—so that reports can be run against the data without disturbing client applications using the original database—is one way data can be replicated.

The following list describes the advantages of data replication:

- Clients accessing replicated data locally—as opposed to remote data that is not replicated—experience improved performance because they do not have to use network services.
- Clients at all sites experience improved availability with replicated data, because if local replicated data is unavailable, a copy of the data is still available, albeit remotely.

These advantages do not come without a cost. Data replication obviously requires more storage for replicated data than for unreplicated data, and updating replicated data can take more processing time than updating a single object.

Data replication can actually be implemented in the logic of client applications, by explicitly specifying where data should be found or updated. However, this way of achieving data replication is costly, error-prone, and difficult to maintain. Instead, the concept of data replication is often coupled with *replication transparency*. Replication transparency is functionality built into a database server (instead of client applications) to automatically handle the details of locating and maintaining data replicas.

INFORMIX-OnLine Dynamic Server Data Replication

Within the broad framework of data replication, **OnLine** implements nearly transparent data replication of entire database servers. All the data managed by one **OnLine** database server is replicated and dynamically updated on another **OnLine** database server, usually at a remote site. **OnLine** data replication is sometimes called *hot site backup*, because it provides a means of maintaining a backup copy of the entire database server that can be used quickly in the event of a catastrophic failure.

Because **OnLine** provides replication transparency, you generally do not need to be concerned with or aware of data replication; the **OnLine** administrator takes care of it. However, if your organization decides to use data replication, you should be aware that special connectivity considerations exist for client applications in a data replication environment. These considerations are described in the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

Summary

Database access is regulated by the privileges that the database owner grants to you. The privileges that let you query data are often granted automatically, but the ability to modify data is regulated by specific Insert, Delete, and Update privileges that are granted on a table-by-table basis.

If data integrity constraints are imposed on the database, your ability to modify data is restricted by those constraints. Your database- and table-level privileges, along with any data constraints, control how and when you can modify data.

You can delete one or more rows from a table with the DELETE statement. Its WHERE clause selects the rows; use a SELECT statement with the same clause to preview the deletes.

Rows are added to a table with the INSERT statement. You can insert a single row containing specified column values, or you can insert a block of rows generated by a SELECT statement.

You use the UPDATE statement to modify the contents of existing rows. You specify the new contents with expressions that can include subqueries, so that you can use data based on other tables or the updated table itself. The statement has two forms: in the first form you specify new values column by column; you use the other form when the new values are generated as a set from a SELECT statement or a record variable.

You use the REFERENCES clause of the CREATE TABLE and ALTER TABLE statements to create relationships between tables. The ON DELETE CASCADE option of the REFERENCES clause allows you to delete rows from parent and associated child tables with one DELETE statement.

You use transactions to prevent unforeseen interruptions in a modification from leaving the database in an indeterminate state. When modifications are performed within a transaction, they are rolled back following an error. The transaction log also extends the periodically made archive copy of the database, so that if the database must be restored, it can be brought back to its most recent state.

Data replication, which is transparent to users, offers another type of protection from catastrophic failures.

Programming with SQL

Chapter Overview	3
SQL in Programs	3
Static Embedding	5
Dynamic Statements	5
Program Variables and Host Variables	5
Calling the Database Server	8
The SQL Communications Area	8
The SQLCODE Field	11
End of Data	11
Negative Codes	11
The SQLERRD Array	12
The SQLAWARN Array	12
The SQLSTATE Value	12
Retrieving Single Rows	13
Data Type Conversion	14
Dealing with Null Data	15
Dealing with Errors	16
End of Data	17
Serious Errors	17
Using Default Values	17
End of Data with Databases that are not ANSI-Compliant	18
Interpreting End of Data with Aggregate Functions	18
Retrieving Multiple Rows	19
Declaring a Cursor	20
Opening a Cursor	20

Fetching Rows	21
Detecting End of Data	21
Locating the INTO Clause	22
Cursor Input Modes	23
The Active Set of a Cursor	24
Creating the Active Set	24
The Active Set for a Sequential Cursor	25
The Active Set for a Scroll Cursor	25
The Active Set and Concurrency	25
Using a Cursor: A Parts Explosion	26
Dynamic SQL	28
Preparing a Statement	29
Executing Prepared SQL	31
Using Prepared SELECT Statements	31
Dynamic Host Variables	32
Freeing Prepared Statements	33
Quick Execution	34
Embedding Data Definition Statements	34
Embedding Grant and Revoke Privileges	34
Summary	37

Chapter Overview

In the examples in the previous chapters, SQL is treated as if it were an interactive computer language; that is, as if you could type a SELECT statement directly into the database server and see rows of data rolling back to you.

Of course, that is not how things are. Many layers of software stand between you and the database server. The database server retains data in a binary form that must be formatted before it can be displayed. It does not return a mass of data at once; it returns one row at a time, as a program requests it.

You can access information in your database in two ways: through interactive access using **DB-Access** or **INFORMIX-SQL** or through application programs written using an SQL API.

Almost any program can contain SQL statements, execute them, and retrieve data from a database server. This chapter explains how these activities are performed and indicates how you can write programs that perform them.

This chapter is only an introduction to the concepts that are common to SQL programming in any language. Before you can write a successful program in a particular programming language, you must first become fluent in that language. Then, because the details of the process are slightly different in every language, you must become familiar with the manual for the Informix SQL API specific to that language.

SQL in Programs

You can write a program in any of several languages and mix SQL statements in among the other statements of the program, just as if they were ordinary statements of that programming language. The SQL statements are said to be *embedded* in the program, and the program is said to contain *embedded SQL*, often abbreviated by Informix as **ESQL**.

Informix produces SQL APIs for the following programming languages:

- C
- COBOL
- FORTRAN (pre-version 6.0)
- Ada (version 4.0 only)

All SQL APIs products work in a similar way, as shown in Figure 5-1. You write a source program in which you treat SQL statements as executable code. Your source program is processed by an embedded SQL *preprocessor*, a program that locates the embedded SQL statements and converts them into a series of procedure calls and special data structures.

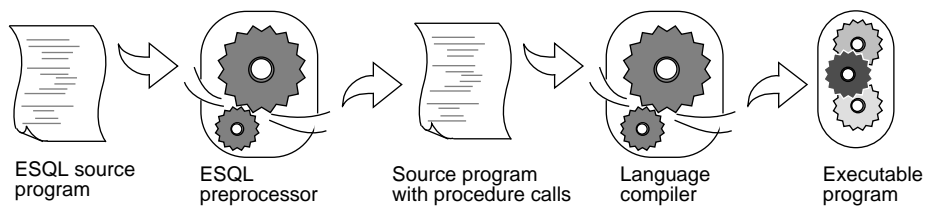


Figure 5-1 Overview of processing a program with embedded SQL statements

The converted source program then passes through the programming language compiler. The compiler output becomes an executable program after it is linked with a library of SQL API procedures. When the program runs, the SQL API library procedures are called; they set up communications with the database server to carry out the SQL operations.

Whereas SQL APIs products allow you to embed SQL in the host language, some languages have SQL as a natural part of their statement set. **INFORMIX-4GL** incorporates the SQL language as a natural part of the fourth-generation language it supports. Informix Stored Procedure Language (SPL) also uses SQL as a natural part of its statement set. You use **INFORMIX-4GL** or an SQL API product to write application programs. You use SPL to write procedures that are stored with a database and are called from an application program.

Static Embedding

You can introduce SQL statements into a program in two ways. The simpler and more common way is by *static embedding*, which means that the SQL statements are written as part of the source program text. The statements are *static* because they are a fixed part of the source text.

Dynamic Statements

Some applications require the ability to compose SQL statements in response to user input. For example, a program might have to select different columns or apply different criteria to rows, depending on what the user wants.

This can be done with *dynamic SQL*, in which the program composes an SQL statement as a string of characters in memory and passes it to the database server to be executed. Dynamic statements are not part of the program source text; they are constructed in memory during execution.

Program Variables and Host Variables

Application programs can use program variables within SQL statements. In **INFORMIX-4GL** and **SPL**, you put the program variable in the SQL statement as syntax allows. For example, a **DELETE** statement can use a program variable in its **WHERE** clause. Figure 5-2 shows a program variable in **INFORMIX-4GL**.

```
MAIN
.
.
.
DEFINE drop_number INT
LET drop_number = 108
DELETE FROM items WHERE order_num = drop_number
.
.
.
```

Figure 5-2 *Using a program variable in **INFORMIX-4GL***

Figure 5-3 shows a program variable in SPL.

```
CREATE PROCEDURE delete_item (drop_number INT)
.
.
.
DELETE FROM items WHERE order_num = drop_number
.
.
.
```

Figure 5-3 *Using a program variable in SPL*

In applications that use embedded SQL statements, the SQL statements can refer to the contents of program variables. A program variable that is named in an embedded SQL statement is called a *host variable* because the SQL statement is thought of as being a “guest” in the program.

The following example is a DELETE statement as it might appear when embedded in a COBOL source program:

```
EXEC SQL
    DELETE FROM items
        WHERE order_num = :o-num
END-EXEC.
```

The first and last lines mark off embedded SQL from the normal COBOL statements. Between them you see an ordinary DELETE statement, as described in Chapter 4, “Modifying Data.” When this part of the COBOL program is executed, a row of the **items** table is deleted; multiple rows can also be deleted.

The statement contains one new feature. It compares the **order_num** column to an item written as **:o-num**, which is the name of a host variable.

Each SQL API product provides a means of delimiting the names of host variables when they appear in the context of an SQL statement. In COBOL, host variable names are designated with an initial colon. The example statement asks the database server to delete rows in which the order number equals the current contents of the host variable named **:o-num**. This is a numeric variable that has been declared and assigned a value earlier in the program.

The same statement embedded in a FORTRAN program looks like the following example:

```
EXEC SQL
  DELETE FROM items
  WHERE order_num = :onum
```

The same statement embedded in an Ada program looks like the following example:

```
EXEC SQL
  DELETE FROM items
  WHERE order_num = $onum;
```

In **INFORMIX-ESQL/Ada**, a host variable is indicated by a leading dollar sign, and statements end with a semicolon. The same is true of **INFORMIX-ESQL/C**, where the corresponding DELETE statement is written as shown in the following example:

```
EXEC SQL delete FROM items
  WHERE order_num = :onum;
```

In **INFORMIX-ESQL/C**, an SQL statement can be introduced with either a leading dollar sign or the words EXEC SQL.

These differences of syntax are trivial; the essential points in all languages (an SQL API, **INFORMIX-4GL**, or SPL) are described in the following list:

- You can embed SQL statements in a source program as if they were executable statements of the host language.
- You can use program variables in SQL expressions the way literal values are used.

If you have programming experience, you can immediately see the possibilities. In the example, the order number to be deleted is passed in the variable **onum**. That value comes from any source that a program can use: it can be read from a file, the program can prompt a user to enter it, or it can be read from the database. The DELETE statement itself can be part of a subroutine (in which case **onum** can be a parameter of the subroutine); the subroutine can be called once or repetitively.

In short, when you embed SQL statements in a program, you can apply all the power of the host language to them. You can hide the SQL under a multitude of interfaces, and you can embellish its functions in a multitude of ways.

Calling the Database Server

Executing an SQL statement is essentially calling the database server as a subroutine. Information must pass from the program to the database server and information must be returned.

Some of this communication is done through host variables. You can think of the host variables named in an SQL statement as the parameters of the procedure call to the database server. In the preceding example, a host variable acts as a parameter of the WHERE clause. Also, as described in “Retrieving Multiple Rows” on page 5-19, host variables receive data that is returned by the database server.

The SQL Communications Area

The database server always returns a result code, and possibly other information about the effect of an operation, in a data structure known as the SQL Communications Area (SQLCA). If the database server executes an SQL statement in a stored procedure, the SQLCA of the calling application contains the values triggered by the SQL statement in the procedure.

The principal fields of the SQLCA are listed in Figure 5-4 and Figure 5-5. The syntax you use to describe a data structure such as the SQLCA, as well as the syntax you use to refer to a field in it, differs among programming languages. See your SQL API manual for details.

You can also use the SQLSTATE variable of the GET DIAGNOSTICS statement to detect, handle, and diagnose errors. See “The SQLSTATE Value” on page 5-12.

integer	SQLCODE	
0	Success.	
100	No more data/not found.	
negative	Error code.	
array of 6 integers	SQLERRD	
first	Following successful prepare of a SELECT, UPDATE, INSERT, or DELETE statement or after a select cursor is opened, this field contains the estimated number of affected rows. It is not used in INFORMIX-ESQL/Ada .	
second	<p>When SQLCODE contains an error code, this field contains either zero or an additional error code, called the ISAM error code, that explains the cause of the main error.</p> <p>After a failed CONNECT or DATABASE statement, this field contains the ISAM, operating system, or network-specific protocol error code.</p> <p>If an application connects to an INFORMIX-Gateway with DRDA, this field contains the application server error code. In this case, sqlca.sqlcode is -29000.</p> <p>Following a successful insert operation of a single row, this field contains the value of a generated serial number for that row.</p>	
third	<p>Following a successful multirow insert, update, or delete operation, this field contains the number of processed rows.</p> <p>Following a multirow insert, update, or delete operation that ends with an error, this field contains the number of rows successfully processed before the error was detected.</p>	
fourth	Following successful prepare of a SELECT, UPDATE, INSERT, or DELETE statement or after a select cursor has been opened, this field contains the estimated weighted sum of disk accesses and total rows processed. It is not used in INFORMIX-ESQL/Ada .	
fifth	Following a syntax error in a PREPARE, EXECUTE IMMEDIATE, DECLARE, or static SQL statement, this field contains the offset in the statement text where the error was detected.	
sixth	Following a successful fetch of a selected row or a successful insert, update, or delete operation, this field contains the rowid (physical address) of the last row processed.	
character (8)	SQLERRP	
	Internal use only.	

Figure 5-4 *The uses of SQLCODE, SQLERRD, and SQLERRP*

array of 8 characters	SQLAWARN
first	When Opening or Connecting to a Database: Set to <i>W</i> when any field is set to <i>W</i> . If this field is blank, the others need not be checked.
second	Set to <i>W</i> when the database now open uses a transaction log.
third	Set to <i>W</i> when the database now open is ANSI-compliant.
fourth	Set to <i>W</i> when the database server is INFORMIX-OnLine Dynamic Server .
fifth	Set to <i>W</i> when the database server stores the FLOAT data type in DECIMAL form (done when the host system lacks support for FLOAT types).
sixth	Not used.
seventh	Set to <i>W</i> when the application is connected to an INFORMIX-OnLine Dynamic Server database server, which is a secondary server in a data replication pair (that is, the database server is available only for read operations).
eighth	Not used.
first	All Other Operations: Set to <i>W</i> when any other field is set to <i>W</i> .
second	Set to <i>W</i> when a column value is truncated as it is fetched into a host variable.
third	Set to <i>W</i> when an aggregate function encounters a null value.
fourth	On a SELECT or on opening a cursor, set to <i>W</i> when the number of items in the select list is not the same as the number of host variables given in the INTO clause to receive them.
fifth	Set to <i>W</i> when any part of a statement involves a host database server which stores the FLOAT data type in DECIMAL form (done when the host system lacks support for FLOAT types).
sixth	Set to <i>W</i> following execution of a statement that does not use ANSI-standard SQL syntax (provided that the DBANSIWARN environment variable is set).
seventh	Not used.
eighth	Not used.
character (71)	SQLERRM
	Contains the variable, such as table name, that is placed in the error message. For some networked applications, contains an error message generated by networking software.

Figure 5-5 The uses of SQLAWARN and SQLERRM

In particular, the subscript by which you name one element of the `SQLERRD` and `SQLWARN` arrays differs: array elements are numbered starting with zero in **INFORMIX-ESQL/C**, but starting with one in the other languages. In this discussion, the fields are named using specific words such as *third*, and you must translate into the syntax of your programming language.

The SQLCODE Field

The `SQLCODE` field is the primary return code of the database server. After every SQL statement, `SQLCODE` is set to an integer value as shown in Figure 5-4. When that value is zero, the statement is performed without error. In particular, when a statement is supposed to return data into a host variable, a code of zero means that the data has been returned and can be used. Any nonzero code means the opposite: no useful data was returned to host variables.

In **INFORMIX-4GL**, `SQLCODE` is also accessible under the name `STATUS`.

End of Data

The database server sets `SQLCODE` to 100 when the statement is performed correctly but no rows are found. Two situations exist where this can occur.

The first situation involves a query that uses a cursor. (Queries using cursors are described under “Retrieving Multiple Rows” on page 5-19.) In these queries the `FETCH` statement retrieves each value from the active set into memory. After the last row is retrieved, a subsequent `FETCH` statement cannot return any data. When this occurs the database server sets `SQLCODE` to 100, which says *end of data-no rows found*.

The second situation involves a query that does not use a cursor. In this case the server sets `SQLCODE` to 100 when no rows satisfy the query condition. In ANSI-compliant databases, `SELECT`, `DELETE`, `UPDATE`, and `INSERT` statements all set `SQLCODE` to 100 if no rows are returned. In databases that are not ANSI-compliant, only a `SELECT` statement that returns no rows causes `SQLCODE` to be set to 100.

Negative Codes

When something unexpected goes wrong during a statement, the database server returns a negative number in `SQLCODE` to explain the problem. The meanings of these codes are documented in the *Informix Error Messages* manual and in the on-line error message file.

The SQLERRD Array

Some error codes that can be reported in SQLCODE reflect general problems, and the database server can set a more detailed code in the second field of SQLERRD. This second code (referred to as the ISAM error) reveals the low-level error encountered by the database server I/O routines or by the underlying operating system.

The integers in the array named SQLERRD are set to different values following different statements. The first and fourth elements of the array are used only in **INFORMIX-4GL**, **INFORMIX-ESQL/C**, **INFORMIX-ESQL/COBOL**, and **INFORMIX-ESQL/FORTRAN**. The fields are used as shown in Figure 5-4 on page 5-9.

These additional details can be very useful. For example, you can use the value in the third field to report how many rows were deleted or updated. When your program prepares an SQL statement entered by the user and an error is found, the value in the fifth field enables you to display the exact point of error to the user. (**DB-Access** and **INFORMIX-SQL** use this feature to position the cursor when you ask to modify a statement after an error.)

The SQLAWARN Array

The eight character fields in the SQLAWARN array are set to either a blank or a W to indicate a variety of special conditions; only six are used. Their meanings depend on the statement just executed.

A set of warning flags appear when a database opens, that is, following a **CONNECT**, **DATABASE** or **CREATE DATABASE** statement. These flags tell you some characteristics of the database as a whole.

A second set of flags appear following any other statement. These flags reflect unusual events that occur during the statement, events that might not be reflected by SQLCODE.

Both sets of SQLAWARN values are summarized in Figure 5-5.

The SQLSTATE Value

Certain Informix products, such as **INFORMIX-ESQL/C** and **INFORMIX-ESQL/COBOL**, support the SQLSTATE value in compliance with X/Open and ANSI SQL standards. The **GET DIAGNOSTICS** statement reads the SQLSTATE value in order to diagnose errors after you run an SQL statement. The database server returns a result code in a five-character string that is

stored in a variable called SQLSTATE. The SQLSTATE error code, or value, tells you the following information about the most recently executed SQL statement:

- If the statement was successful
- If the statement was successful but generated warnings
- If the statement was successful but generated no data
- If the statement failed

For more information on GET DIAGNOSTICS, the SQLSTATE variable, and the meanings of the SQLSTATE return codes, see the section called “GET DIAGNOSTICS” in Chapter 1 of the *Informix Guide to SQL: Syntax*. If your Informix product supports GET DIAGNOSTICS and SQLSTATE, Informix recommends that you use them as the primary structure to detect, handle, and diagnose errors.

Retrieving Single Rows

You can use embedded SELECT statements to retrieve single rows from the database into host variables. When a SELECT statement returns more than one row of data, however, a program must use a more complicated method to fetch the rows one at a time. Multiple-row select operations are discussed in “Retrieving Multiple Rows” on page 5-19.

To retrieve a single row of data, simply embed a SELECT statement in your program. The following example shows how it can be written using **INFORMIX-ESQL/C**:

```
exec sql select avg (total_price)
into :avg_price
from items
where order_num in
(select order_num from orders
where order_date < date('6/1/93'));
```

The INTO clause is the only detail that distinguishes this statement from any example in Chapter 2, “Composing Simple SELECT Statements,” or Chapter 3, “Composing Advanced SELECT Statements.” This clause specifies the host variables that are to receive the data that is produced.

When the program executes an embedded SELECT statement, the database server performs the query. The example statement selects an aggregate value, so that it produces exactly one row of data. The row has only a single column, and its value is deposited in the host variable named **avg_price**. Subsequent lines of the program can use that variable.

You can use statements of this kind to retrieve single rows of data into host variables. The single row can have as many columns as desired. In the following **INFORMIX-4GL** example, host variables are used in two ways, as receivers of data and in the WHERE clause:

```
DEFINE cfname, clname, ccompany CHAR(20)
DEFINE cnumbr INTEGER
LET cnumbr = 104
SELECT fname, lname, company
    INTO cfname, clname, ccompany
    FROM customer
    WHERE customer_num = cnumbr
```

Because the **customer_num** column has a unique index (implemented through a constraint), this query returns only one row. If a query produces more than one row of data, the database server cannot return any data. It returns an error code instead.

You should list as many host variables in the INTO clause as there are items in the select list. If, by accident, these lists are of different lengths, the database server returns as many values as it can and sets the warning flag in the fourth field of SQLAWARN.

Data Type Conversion

The following example retrieves the average of a DECIMAL column, which is itself a DECIMAL value. However, the host variable into which it is placed is *not* required to have that data type.

```
exec sql select avg (total_price) into :avg_price
    from items;
```

The declaration of the receiving variable **avg_price** in the previous example of **ESQL/C** code is not shown. It could be any one of the following definitions:

```
int avg_price;
double avg_price;
char avg_price[16];
dec_t avg_price; /* typedef of decimal number structure */
```

The data type of each host variable used in a statement is noted and passed to the database server along with the statement. The database server does its best to convert column data into the form used by the receiving variables. Almost any conversion is allowed, although some conversions cause a loss of precision. The results of the preceding example differ depending on the data type of the receiving host variable, as described in the following list:

FLOAT	<p>The database server converts the decimal result to FLOAT, possibly truncating some fractional digits.</p> <p>If the magnitude of a decimal exceeds the maximum magnitude of the FLOAT format, an error is returned.</p>
INTEGER	<p>The database server converts the result to integer, truncating fractional digits if necessary.</p> <p>If the integer part of the converted number does not fit the receiving variable, an error occurs.</p>
CHARACTER	<p>The database server converts the decimal value to a character string.</p> <p>If the string is too long for the receiving variable, it is truncated. The second field of SQLAWARN is set to W and the value in the SQLSTATE variable is 01000.</p>

Dealing with Null Data

What if the program retrieves a null value? Null values can be stored in the database, but the data types supported by programming languages do not recognize a null state. A program must have some way of recognizing a null item to avoid processing it as data.

Indicator variables meet this need in the SQL APIs. An indicator variable is an additional variable that is associated with a host variable that might receive a null item. When the database server puts data in the main variable, it also

puts a special value in the indicator variable to show whether the data is null. In the following **INFORMIX-ESQL/C** example, a single row is selected and a single value is retrieved into the host variable **op_date**:

```
exec sql select paid_date
         into :op_date:op_d_ind
         from orders
         where order_num = $the_order;
if (op_d_ind < 0) /* data was null */
    rstrdate ('01/01/1900', :op_date);
```

Because the value might be null, an indicator variable named **op_d_ind** is associated with the host variable. (It must be declared as a short integer elsewhere in the program.)

Following execution of the **SELECT** statement, the program tests the indicator variable for a negative value. A negative number (usually -1) means that the value retrieved into the main variable is null. If that is the case, this program uses a C function to assign a default value to the host variable. (The function **rstrdate** is part of the **INFORMIX-ESQL/C** product.)

The syntax you use to associate an indicator variable differs with the language you are using, but the principle is the same in all. However, indicator variables are not used explicitly in **INFORMIX-4GL** or in **SPL** because in those languages, null values are supported for variables. In **4GL**, the preceding example is written as follows:

```
SELECT paid_date
      INTO op_date
      FROM orders
      WHERE order_num = the_order
IF op_date IS NULL THEN
    LET op_date = date ('01/01/1900')
END IF
```

Dealing with Errors

Although the database server handles conversion between data types automatically, several things still can go wrong with a **SELECT** statement. In SQL programming, as in any kind of programming, you must anticipate errors and provide for them at every point.

End of Data

One common event is that no rows satisfy a query. This is signalled by an SQLSTATE code of 02000 and by a code of 100 in SQLCODE (`sqlca.sqlcode`, as it is known in **ESQL/C**) following a SELECT statement. This code indicates an error or a normal event, depending entirely on your application. If you are sure that there ought to be a row or rows—for example, if you are reading a row using a key value that you just read from a row of another table—then the end-of-data code represents a serious failure in the logic of the program. On the other hand, if you select a row based on a key supplied by a user or some other source that is less reliable than a program, a lack of data can be a normal event.

Serious Errors

Errors that set SQLCODE to a negative value are usually serious. Programs which you have developed and are in production should rarely report these errors. Nevertheless, it is difficult to anticipate every problematic situation, so your program must be able to deal with them.

For example, a query can return error -206, which means `table name` is not in the database. This happens if someone dropped the table since the program was written or if, through some error of logic or mistake in input, the program opened the wrong database.

Using Default Values

You can handle these inevitable errors in many ways. In some applications, more lines of code to handle errors are used than for normal situations. In the examples in this section, however, one of the simplest methods, the default value, should work.

```
avg_price = 0; /* set default for errors */
exec sql select avg (total_price)
           into :avg_price:null_flag
           from items;
if (null_flag < 0) /* probably no rows */
    avg_price = 0; /* set default for 0 rows */
```

End of Data with Databases that are not ANSI-Compliant

If your database is not ANSI-compliant, the end-of-data return code, 100, is set in SQLCODE only following SELECT statements. In addition, the SQLSTATE value is set to 02000. (Other statements, such as INSERT, UPDATE, and DELETE, set the third element of SQLERRD to show how many rows they affected; this topic is covered in Chapter 6, “Modifying Data Through SQL Programs.”)

Interpreting End of Data with Aggregate Functions

A SELECT statement that selects an aggregate function such as SUM, MIN, or AVG always succeeds in returning at least one row of data. This is true even when no rows satisfy the WHERE clause; an aggregate value based on an empty set of rows is null, but it exists nonetheless.

However, an aggregate value is also null if it is based on one or more rows that all contain null values. If you must be able to detect the difference between an aggregate value based on no rows and one based on some rows that are all null, you must include a COUNT function in the statement and an indicator variable on the aggregate value. You can then work out the following three cases:

Count Value	Indicator	Case
0	-1	zero rows selected
>0	-1	some rows selected; all were null
>0	0	some non-null rows selected

The previous example deals with the following considerations:

- If the query selects some non-null rows, the correct value is returned and used. This is the expected and most frequent result.
- If the query selects no rows, or in the much less likely event that it selects only rows that have null values in the **total_price** column (a column that should never be null), the indicator variable is set and the default value is assigned.
- If any serious error occurs, the host variable is left unchanged; it contains the default value initially set. At this point in the program, the programmer sees no need to trap such errors and report them.

The following example is an expansion of an earlier **INFORMIX-4GL** example that displays default values if it cannot find the company that the user requests:

```
DEFINE cfname, clname, ccompany CHAR(20)
DEFINE cnumbr INTEGER
PROMPT 'Enter the customer number: ' FOR cnumbr
LET cfname = 'unknown'
LET clname = 'person'
LET ccompany = 'noplac'
SELECT fname, lname, company
      INTO cfname, clname, ccompany
      WHERE customer_num = cnumbr
DISPLAY cfname, ' ', clname, ' at ', ccompany
```

This query does not use aggregates, so if no row matches the user-specified customer number, **SQLCODE** is set to 100 and the host variables remain unchanged.

Retrieving Multiple Rows

When any chance exists that a query could return more than one row, the program must execute the query differently. Multirow queries are handled in two stages. First, the program starts the query. (No data is returned immediately.) Then, the program requests the rows of data one at a time.

These operations are performed using a special data object called a *cursor*. A cursor is a data structure that represents the current state of a query. The following list is the general sequence of program operations:

1. The program *declares* the cursor and its associated **SELECT** statement. This merely allocates storage to hold the cursor.
2. The program *opens* the cursor. This starts the execution of the associated **SELECT** statement and detects any errors in it.
3. The program *fetches* a row of data into host variables and processes it.
4. The program *closes* the cursor after the last row is fetched.

These operations are performed with SQL statements named **DECLARE**, **OPEN**, **FETCH**, and **CLOSE**.

Declaring a Cursor

You declare a cursor using the DECLARE statement. This statement gives the cursor a name, specifies its use, and associates it with a statement. The following simple example is written in **INFORMIX-4GL**:

```
DECLARE the_item CURSOR FOR
  SELECT order_num, item_num, stock_num
  INTO o_num, i_num, s_num
  FROM items
```

The declaration gives the cursor a name (**the_item** in this case) and associates it with a SELECT statement. (Chapter 6, “Modifying Data Through SQL Programs,” discusses how a cursor also can be associated with an INSERT statement.)

The SELECT statement in this example contains an INTO clause. That is one of two ways in which you can specify the variables that receive data. The other way, using the FETCH statement, is discussed in “Locating the INTO Clause” on page 5-22.

The DECLARE statement is not an active statement; it merely establishes the features of the cursor and allocates storage for it. You can use the cursor declared in the preceding example to read once through the **items** table. Cursors can be declared to read both backward and forward (see “Cursor Input Modes” on page 5-23). This cursor, because it lacks a FOR UPDATE clause, probably is used only to read data, not to modify it. (The use of cursors to modify data is covered in Chapter 6, “Modifying Data Through SQL Programs.”)

Opening a Cursor

The program opens the cursor when it is ready to use it. The OPEN statement activates the cursor. It passes the associated SELECT statement to the database server, which begins the search for matching rows. The database server processes the query to the point of locating or constructing the first row of output. It does not actually return that row of data, but it does set a return code in SQLSTATE for SQL APIs and SQLCODE for **INFORMIX-4GL** and SQL APIs. The following example shows the OPEN statement in **INFORMIX-4GL**:

```
OPEN the_item
```

Because this is the first time that the database server has seen the query, it is the time when many errors are detected. After opening the cursor, the program should test SQLSTATE or SQLCODE. If the SQLSTATE value is greater than 02000 or the SQLCODE contains a negative number, the cursor is not usable. There may be an error in the SELECT statement, or some other problem may be preventing the database server from executing the statement.

If SQLSTATE is equal to 00000 or SQLCODE contains a zero, the SELECT statement is syntactically valid, and the cursor is ready for use. At this point, however, the program does not know if the cursor can produce any rows.

Fetching Rows

The program uses the FETCH statement to retrieve each row of output. This statement names a cursor, and also can name the host variables to receive the data. Here is the **INFORMIX-4GL** example completed:

```
DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
        INTO o_num, i_num, s_num
        FROM items
OPEN the_item
WHILE sqlcode = 0
    FETCH the_item
    IF sqlcode = 0 THEN
        DISPLAY o_num, i_num, s_num
    END IF
END WHILE
```

Detecting End of Data

In the previous example, the WHILE condition prevents execution of the loop in case the OPEN statement returns an error. The same condition terminates the loop when SQLCODE is set to 100 to signal the end of data. However, the loop contains a test of SQLCODE. This test is necessary because, if the SELECT

statement is valid yet finds no matching rows, the OPEN statement returns a zero but the first fetch returns 100, end of data, and no data. The following example shows another way to write the same loop:

```
DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
        INTO o_num, i_num, s_num
        FROM items
OPEN the_item
IF sqlcode = 0 THEN
    FETCH the_item -- fetch first row
END IF
WHILE sqlcode = 0
    DISPLAY o_num, i_num, s_num
    FETCH the_item
END WHILE
```

In this version, the case of zero returned rows is handled early, so no second test of **sqlcode** exists within the loop. These versions have no measurable difference in performance because the time cost of a test of **sqlcode** is a tiny fraction of the cost of a fetch.

Locating the INTO Clause

The INTO clause names the host variables that are to receive the data returned by the database server. It must appear in either the SELECT or the FETCH statement but not both. The following example is reworked to specify host variables in the FETCH statement:

```
DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
        FROM items
OPEN the_item
WHILE status = 0
    FETCH the_item INTO o_num, i_num, s_num
    IF status = 0 THEN
        DISPLAY o_num, i_num, s_num
    END IF
END WHILE
```

The second form has the advantage that different rows can be fetched into different variables. For example, you can use this form to fetch successive rows into successive elements of an array.

Cursor Input Modes

For purposes of input, a cursor operates in one of two modes: *sequential* or *scrolling*. A sequential cursor can fetch only the next row in sequence so it can read through a table only once each time it is opened. A scroll cursor can fetch the next row or any prior row, so it can read rows multiple times. The following example shows a sequential cursor declared in **INFORMIX-ESQL/C**:

```
EXEC SQL DECLARE pcurs CURSOR FOR
      SELECT customer_num, lname, city
      FROM customer;
```

After it is opened, the cursor can be used only with a sequential fetch that retrieves the next row of data, as shown in the following example:

```
EXEC SQL FETCH p_curs INTO :cnum, :clname, :ccity;
```

Each sequential fetch returns a new row.

A scroll cursor is declared with the keyword **SCROLL**, as shown in the following example from **INFORMIX-ESQL/FORTRAN**:

```
EXEC SQL DECLARE s_curs SCROLL CURSOR FOR
+   SELECT order_num, order_date FROM orders
+   WHERE customer_num > 104
```

A scroll cursor can be used with a variety of fetch options. The **ABSOLUTE** option specifies the rank number of the row to fetch.

```
EXEC SQL FETCH ABSOLUTE numrow s_curs
+   INTO :nordr, :nodat
```

This statement fetches the row whose position is given in the host variable **numrow**. It is also possible to fetch the current row again or to fetch the first row and then scan through the entire list again. However, these features are obtained at a price, as described in the next section.

The Active Set of a Cursor

Once a cursor is opened, it stands for some selection of rows. The set of all rows that the query produces is called the *active set* of the cursor. It is easy to think of the active set as a well-defined collection of rows and to think of the cursor as pointing to one row of the collection. This situation is true as long as no other programs are modifying the same data concurrently.

Creating the Active Set

When a cursor is opened, the database server does whatever is necessary to locate the first row of selected data. Depending on how the query is phrased, this can be very easy to do, or it can require a great deal of work and time.

```
DECLARE easy CURSOR FOR
  SELECT fname, lname FROM customer
  WHERE state = 'NJ'
```

Because this cursor queries only a single table in a simple way, the database server can very quickly discover whether any rows satisfy the query and find the first one. The first row is the only row it finds at this time. The rest of the rows in the active set remain unknown.

```
DECLARE hard CURSOR FOR
  SELECT C.customer_num, O.order_num, sum (items.total_price)
  FROM customer C, orders O, items I
  WHERE C.customer_num = O.customer_num
        AND O.order_num = I.order_num
        AND O.paid_date is null
  GROUP BY C.customer_num, O.order_num
```

The active set of this cursor is generated by joining three tables and grouping the output rows. The optimizer might be able to use indexes to produce the rows in the correct order, but in general the use of ORDER BY or GROUP BY clauses requires the database server to generate all the rows, copy them to a temporary table, and sort the table, before it can know which row to present first. This is discussed in Chapter 13, “Optimizing Your Queries.”

In cases where the active set is entirely generated and saved in a temporary table, the database server can take quite some time to open the cursor. Afterward, it can tell the program exactly how many rows the active set contains. This information is not made available, however. One reason is that you can

never be sure which method the optimizer uses. If it can avoid sorts and temporary tables, it does; but very small changes in the query, in the sizes of the tables, or in the available indexes can change its methods.

The Active Set for a Sequential Cursor

The database server wants to tie up as few resources as possible in maintaining the active set of a cursor. If it can do so, the database server never retains more than the single row that is fetched next. It can do this for most sequential cursors. On each fetch, it returns the contents of the current row and locates the next one.

The Active Set for a Scroll Cursor

All the rows in the active set for a scroll cursor must be retained until the cursor closes because the database server cannot be sure which row the program will ask for next.

Most frequently, the database server implements the active set of a scroll cursor as a temporary table. The database server might not fill this table immediately, however (unless it created a temporary table to process the query). Usually it creates the temporary table when the cursor is opened. Then, the first time a row is fetched, the database server copies it into the temporary table and returns it to the program. When a row is fetched for a second time, it can be taken from the temporary table. This scheme uses the fewest resources in the event that the program abandons the query before it fetches all the rows. Rows that are never fetched are not created or saved.

The Active Set and Concurrency

When only one program is using a database, the members of the active set cannot change. This is the situation in most desktop workstations, and it is the easiest situation to think about. But some programs must be designed for use in a multiprogramming system, where two, three, or dozens of different programs can work on the same tables simultaneously.

When other programs can update the tables while your cursor is open, the idea of the active set becomes less useful. Your program can see only one row of data at a time, but all other rows in the table can be changing.

In the case of a simple query, when the database server holds only one row of the active set, any other row can change. The instant after your program fetches a row, another program can delete the same row or update it so that if it is examined again, it is no longer part of the active set.

When the active set, or part of it, is saved in a temporary table, *stale data* can present a problem. That is, the rows in the actual tables, from which the active-set rows are derived, can change. If they do, some of the active-set rows no longer reflect the current table contents.

These ideas may seem unsettling at first, but as long as your program only reads the data, there is no such thing as stale data, or rather, all data is equally stale. No matter when it is taken, the active set is a snapshot of the data as it is at one moment in time. A row is different the next day; it does not matter if it is also different in the next millisecond. To put it another way, no practical difference exists between changes that occur while the program is running and changes that are saved and applied the instant the program terminates.

The only time that stale data can cause a problem is when the program intends to use the input data to modify the same database; for example, when a banking application must read an account balance, change it, and write it back. Chapter 6, “Modifying Data Through SQL Programs,” discusses programs that modify data.

Using a Cursor: A Parts Explosion

When you use a cursor, supplemented by program logic, you can solve problems that plain SQL cannot solve. One of these is the parts-explosion problem, sometimes called Bill of Materials processing. At the heart of this problem is a recursive relationship among objects; one object contains other objects, which contain yet others.

The problem is usually stated in terms of a manufacturing inventory. A company makes a variety of parts, for example. Some parts are discrete, but some are assemblages of other parts.

These relationships are documented in a single table, which might be called **contains**. The column **contains.parent** holds the part numbers of parts that are assemblages. The column **contains.child** has the part number of a part that is a component of the parent. If part #123400 is an assembly of nine parts, nine rows exist with 123400 in the first column and other part numbers in the second.

CONTAINS

PARENT	CHILD	
FK NN	FK NN	
123400	432100	
432100	765899	

The parts-explosion problem is this: given a part number, produce a list of all parts that are components of that part. A sketch of one solution, as implemented in **INFORMIX-4GL**, is shown in Figure 5-6.

```

DEFINE part_list ARRAY[200] OF INTEGER
FUNCTION boom (top_part)
  DEFINE this_part, child_part INTEGER
  DEFINE next_to_do, next_free SMALLINT
  DECLARE part_scan CURSOR FOR
    SELECT child INTO child_part FROM contains
      WHERE parent = this_part

  LET next_to_do = 1
  LET part_list[next_to_do] = top_part
  LET next_free = 2

  WHILE next_to_do < next_free
    this_part = part_list[next_to_do]
    FOREACH part_scan
      LET part_list[next_free] = child_part
      LET next_free = next_free + 1
    END FOREACH
    LET next_to_do = next_to_do + 1
  END WHILE
  RETURN next_free - 1
END FUNCTION

```

Figure 5-6 *A breadth-first algorithm to generate a parts explosion*

Technically speaking, each row of the **contains** table is the head node of a directed acyclic graph, or *tree*. The function in Figure 5-6 performs a breadth-first search of the tree whose root is the part number passed as its parameter. The function uses a cursor named **part_scan** to return all the rows with a particular value in the **parent** column. This is very easy to implement using the **INFORMIX-4GL** statement **FOREACH**, which opens a cursor, iterates once for each row in the selection set, and closes the cursor.

This is the heart of the parts-explosion problem, but it is not a complete solution. For example, the program in Figure 5-6 does not allow for components that appear at more than one level in the tree. Furthermore, a practical **contains** table would also have a column **count**, giving the count of **child** parts used in each **parent**. A program that returns a total count of each component part is much more complicated.

The iterative approach described earlier is not the only way to approach the parts-explosion problem. If the number of generations has a fixed limit, you can solve the problem with a single SELECT statement using nested, outer self-joins.

If there can be up to four generations of parts contained within one top-level part, the following SELECT statement returns all of them:

```
SELECT a.parent, a.child, b.child, c.child, d.child
FROM contains a
      OUTER (contains b,
              OUTER (contains c, outer contains d))
WHERE a.parent = top_part_number
      AND a.child = b.parent
      AND b.child = c.parent
      AND c.child = d.parent
```

This SELECT statement returns one row for each line of descent rooted in the part given as **top_part_number**. Null values are returned for levels that do not exist. (Use indicator variables to detect them.) You can extend this solution to more levels by selecting additional nested outer joins of the **contains** table. You also can revise this solution to return counts of the number of parts at each level.

Dynamic SQL

Although static SQL is extremely useful, it requires that you know the exact content of every SQL statement at the time you write the program. For example, you must state exactly which columns are tested in any WHERE clause and exactly which columns are named in any select list.

This is no problem when you write a program to perform a well-defined task. But the database tasks of some programs cannot be perfectly defined in advance. In particular, a program that must respond to an interactive user might need the ability to compose SQL statements in response to what the user enters.

Dynamic SQL allows a program to form an SQL statement during execution, so that the contents of the statement can be determined by user input. This is done in the following steps:

1. The program assembles the text of an SQL statement as a character string, which is stored in a program variable.
2. It executes a PREPARE statement, which asks the database server to examine the statement text and prepare it for execution.
3. It uses the EXECUTE statement to execute the prepared statement.

In this way, a program can construct and then use any SQL statement, based on user input of any kind. For example, it can read a file of SQL statements and prepare and execute each.

DB-Access, the utility that you use to explore SQL interactively, is an **INFORMIX-ESQL/C** program that constructs, prepares, and executes SQL statements dynamically. For example, it enables users to specify the columns of a table using simple, interactive menus. When the user is finished, **DB-Access** builds the necessary CREATE TABLE or ALTER TABLE statement dynamically and prepares and executes it.

Preparing a Statement

In form, a dynamic SQL statement is like any other SQL statement that is written into a program, except that it cannot contain the names of any host variables.

This leads to two restrictions. First, if it is a SELECT statement, it cannot include the INTO clause. The INTO clause names host variables into which column data is placed, and host variables are not allowed in a dynamic statement. Second, wherever the name of a host variable normally appears in an expression, a question mark is written as a placeholder.

You can prepare a statement in this form for execution with the PREPARE statement. The following example is in **INFORMIX-ESQL/C**:

```
exec sql prepare query_2 from
      'select * from orders
       where customer_num = ? and
        order_date > ?';
```

Notice the two question marks in this example; they indicate that when the statement is executed, the values of host variables are used at those two points.

You can prepare almost any SQL statement dynamically. The only ones that cannot be prepared are the ones directly concerned with dynamic SQL and cursor management, such as the PREPARE and OPEN statements. After you prepare an UPDATE or DELETE statement, it is a good idea to test the fifth field of SQLAWARN to see if you used a WHERE clause (see “The SQLAWARN Array” on page 5-12).

The result of preparing a statement is a data structure that represents the statement. This data structure is not the same as the string of characters that produced it. In the PREPARE statement, you give a name to the data structure; it is **query_2** in the preceding example. This name is used to execute the prepared SQL statement.

The PREPARE statement does not limit the character string to one statement. It may contain multiple SQL statements, separated by semicolons. Figure 5-7 shows a fairly complex example in **INFORMIX-ESQL/COBOL**:

```
MOVE      'BEGIN WORK;
          UPDATE account
            SET balance = balance + ?
            WHERE acct_number = ?;
          UPDATE teller
            SET balance = balance + ?
            WHERE teller_number = ?;
          UPDATE branch
            SET balance = balance + ?
            WHERE branch_number = ?;
          INSERT INTO history VALUES(timestamp, values);

          TO BIG-QUERY.

EXEC SQL
          PREPARE BIG-Q FROM :BIG-QUERY
END-EXEC.
```

Figure 5-7 *Preparing a string containing five SQL statements*

When this list of statements is executed, host variables must provide values for six place-holding question marks. Although it is more complicated to set up a multistatement list, the performance is often better because fewer exchanges take place between the program and the database server.

Executing Prepared SQL

Once a statement is prepared, it can be executed multiple times. Statements other than SELECT statements, and SELECT statements that return only a single row, are executed with the EXECUTE statement.

Figure 5-8 shows how INFORMIX-ESQL/C prepares and executes a multistatement update of a bank account:

```
exec sql begin declare section;
char bigquery[270] = 'begin work;';
exec sql end declare section;
stcat ('update account set balance = balance + ? where ', bigquery);
stcat ('acct_number = ?;', bigquery);
stcat ('update teller set balance = balance + ? where ', bigquery);
stcat ('teller_number = ?;', bigquery);
stcat ('update branch set balance = balance + ? where ', bigquery);
stcat ('branch_number = ?;', bigquery);
stcat ('insert into history values(timestamp, values);', bigquery);

exec sql prepare bigq from :bigquery;

exec sql execute bigq using :delta, :acct_number, :delta,
                           :teller_number, :delta, :branch_number;

exec sql commit work;
```

Figure 5-8 *Preparing and executing a multistatement operation in ESQL/C*

The USING clause of the EXECUTE statement supplies a list of host variables whose values are to take the place of the question marks in the prepared statement.

Using Prepared SELECT Statements

A dynamically prepared SELECT statement cannot simply be executed; it might produce more than one row of data, and the database server, not knowing which row to return, produces an error code.

Instead, a dynamic SELECT statement is attached to a cursor. Then, the cursor is opened and used in the usual way. The cursor to be used with a prepared statement is declared for that statement name. The following example is in **INFORMIX-4GL**:

```
LET select_2 = 'select order_num, order_date from orders ',
               'where customer_num = ? and order_date > ?'

PREPARE q_orders FROM select_2

DECLARE cu_orders CURSOR FOR q_orders

OPEN cu_orders USING q_c_number, q_o_date

FETCH cu_orders INTO f_o_num, f_o_date
```

The following list identifies the stages of processing in this example:

1. A character string expressing a SELECT statement is placed in a program variable. It employs two place-holding question marks.
2. The PREPARE statement converts the string into a data structure that can be executed. The data structure is associated with a name, **q_orders**.
3. A cursor named **cu_orders** is declared and associated with the name of the prepared statement.
4. When the cursor is opened, the prepared statement is executed. The USING clause in the OPEN statement provides the names of two host variables whose contents are substituted for the question marks in the original statement.
5. The first row of data is fetched from the open cursor. The INTO clause of the FETCH statement specifies the host variables that are to receive the fetched column values.

Later, the cursor can be closed and reopened. While the cursor is closed, a different SELECT statement can be prepared under the name **q_orders**. In this way, a single cursor can be used to fetch from different SELECT statements.

Dynamic Host Variables

In SQL APIs, which support dynamically allocated data objects, it is possible to take dynamic statements one step further. It is possible to dynamically allocate the host variables that receive column data.

This makes it possible to take an arbitrary SELECT statement from program input, determine how many values it produces and their data types, and allocate the host variables of the appropriate types to hold them.

The key to this ability is the DESCRIBE statement. It takes the name of a prepared SQL statement and returns information about the statement and its contents. It sets SQLCODE to specify the type of statement, that is, the verb with which it begins. If the prepared statement is a SELECT, the DESCRIBE statement also returns information about the selected output data. If the prepared statement is an INSERT, the DESCRIBE statement returns information about the input parameters. The data structure is a predefined data structure allocated for this purpose and is known as a system descriptor area. If you are using INFORMIX-ESQL/C, you can use a system descriptor area or, as an alternative, an **sqllda** structure.

The data structure that a DESCRIBE statement returns or references for a SELECT statement includes an array of structures. Each structure describes the data that is returned for one item in the select list. The program can examine the array and discover that a row of data includes a decimal value, a character value of a certain length, and an integer.

Using this information, the program can allocate memory to hold the retrieved values and put the necessary pointers in the data structure for the database server to use.

Freeing Prepared Statements

A prepared SQL statement occupies space in memory. With some database servers, it can consume space owned by the database server as well as space belonging to the program. This space is released when the program terminates, but in general, you should free this space when you finish with it.

You can use the FREE statement to release this space. It takes either the name of a statement or the name of a cursor that was declared FOR a statement name, and releases the space allocated to the prepared statement. If more than one cursor is defined on the statement, freeing the statement does not free the cursor.

Quick Execution

For simple statements that do not require a cursor or host variables, you can combine the actions of the PREPARE, EXECUTE, and FREE statements into a single operation. The EXECUTE IMMEDIATE statement, shown in the following example, takes a character string and in one operation prepares it, executes it, and frees the storage.

```
exec sql execute immediate 'drop index my_temp_index';
```

This makes it easy to write simple SQL operations. However, because no USING clause is allowed, the EXECUTE IMMEDIATE statement cannot be used for SELECT statements.

Embedding Data Definition Statements

Data definition statements, the SQL statements that create databases and modify the definitions of tables, are not usually put into programs. The reason is that they are rarely performed—a database is created once, but is queried and updated many times.

The creation of a database and its tables is generally done interactively, using **DB-Access** or **INFORMIX-SQL**. These tools can also be driven from a file of statements, so that the creation of a database can be done with one operating system command. (This is discussed in Chapter 10, “Tuning Your Data Model.”)

Embedding Grant and Revoke Privileges

One task related to data definition is done repeatedly: the granting and revoking of privileges. The reasons for this are discussed in Chapter 11, “Granting and Limiting Access to Your Database.” Because privileges must be granted and revoked frequently, and because this might be done by persons who are not skilled in SQL, it can be useful to package the GRANT and REVOKE statements in programs to give them a simpler, more convenient user interface.

The GRANT and REVOKE statements are especially good candidates for dynamic SQL. Each statement takes the following parameters:

- A list of one or more privileges
- A table name
- The name of a user

You probably need to supply at least some of these values based on program input (from the user, command-line parameters, or a file) but none can be supplied in the form of a host variable. The syntax of these statements does not allow host variables at any point.

The only alternative is to assemble the parts of a statement into a character string, and prepare and execute the assembled statement. Program input can be incorporated into the prepared statement as characters.

Figure 5-9 shows a function in **INFORMIX-4GL** that assembles a GRANT statement from the function parameters, and then prepares and executes it.

```
FUNCTION table_grant (priv_to_grant, table_name, user_id)
  DEFINE  priv_to_grant char(100),
          table_name char(20),
          user_id char(20),
          grant_stmt char(200)
  LET grant_stmt = ' GRANT ', priv_to_grant,
                  ' ON ', table_name,
                  ' TO ', user_id
  WHENEVER ERROR CONTINUE
  PREPARE the_grant FROM grant_stmt
  IF status = 0 THEN
    EXECUTE the_grant
  END IF
  IF status <> 0 THEN
    DISPLAY 'Sorry, got error #', status, 'attempting:'
    DISPLAY '      ', grant_stmt
  END IF
  FREE the_grant
  WHENEVER ERROR STOP
END FUNCTION
```

Figure 5-9 *A 4GL function that builds, prepares, and executes a GRANT statement*

The opening statement, shown in the following example, defines the name of the function and the names of its three parameters:

```
FUNCTION table_grant (priv_to_grant, table_name, user_id)
```

The **DEFINE** statement, shown in the following example, defines the parameters and one additional variable local to the function. All four are character strings of various lengths.

```
DEFINE  priv_to_grant char(100),
        table_name char(20),
        user_id char(20),
        grant_stmt char(200)
```

The variable **grant_stmt** holds the assembled GRANT statement, which is created by concatenating the parameters and some constants. This is shown in the following example:

```
LET grant_stmt = 'GRANT ', priv_to_grant,
                ' ON ', table_name,
                ' TO ', user_id
```

In **INFORMIX-4GL**, the comma is used to concatenate strings. This assignment statement concatenates the following six character strings:

- 'GRANT'
- The parameter specifying the privileges to be granted
- 'ON'
- The parameter specifying the table name
- 'TO'
- The parameter specifying the user.

The result is a complete GRANT statement composed partly of program input. The same feat can be accomplished in other host languages using different syntax, as shown in the following figure:

```
WHenever Error Continue
PREPARE the_grant FROM grant_stmt
```

If the database server returns an error code in **SQLCODE**, the default action of an **INFORMIX-4GL** program is to terminate. However, errors are quite likely when you prepare an SQL statement composed of user-supplied parts, and program termination is a poor way to diagnose the error. In the preceding code, the **WHENEVER** statement prevents termination. Then the **PREPARE** statement passes the assembled statement text to the database server for parsing.

If the database server approves the form of the statement, it sets a zero return code. This does not guarantee that the statement is executed properly; it means only that the statement has correct syntax. It might refer to a nonexistent table or contain many other kinds of errors that can only be detected during execution.

```
IF status = 0 THEN
    EXECUTE the_grant
END IF
```

If the preparation is successful, the next step is to execute the prepared statement. The function in Figure 5-9 displays an error message if anything goes wrong. As written, it makes no distinction between an error from the PREPARE operation and one from the EXECUTE operation. It does not attempt to interpret the numeric error code, leaving it to the user to interpret.

Summary

SQL statements can be written into programs as if they were normal statements of the programming language. Program variables can be used in WHERE clauses, and data from the database can be fetched into them. A preprocessor translates the SQL code into procedure calls and data structures.

Statements that do not return data, or queries that return only one row of data, are written like ordinary imperative statements of the language. Queries that can return more than one row are associated with a cursor that represents the current row of data. Through the cursor, the program can fetch each row of data as it is needed.

Static SQL statements are written into the text of the program. However, the program can form new SQL statements dynamically, as it runs, and execute them also. In the most advanced cases, the program can obtain information about the number and types of columns that a query returns and dynamically allocate the memory space to hold them.

Summary

Modifying Data Through SQL Programs

Chapter Overview	3
Using DELETE	3
Direct Deletions	4
Errors During Direct Deletions	4
Using Transaction Logging	5
Coordinated Deletions	6
Deleting with a Cursor	7
Using INSERT	8
Using an Insert Cursor	8
Declaring an Insert Cursor	9
Inserting with a Cursor	10
Status Codes After PUT and FLUSH	11
Rows of Constants	11
An Insert Example	12
Using UPDATE	14
Using an Update Cursor	15
The Purpose of the Keyword UPDATE	15
Updating Specific Columns	16
UPDATE Keyword Not Always Needed	16
Cleaning up a Table	16
Summary	17

Chapter Overview

The preceding chapter introduced the idea of putting SQL statements, especially the SELECT statement, into programs written in other languages. This enables a program to retrieve rows of data from a database.

This chapter covers the issues that arise when a program needs to modify the database by deleting, inserting, or updating rows. As in Chapter 5, “Programming with SQL,” this chapter aims to prepare you for reading the manual for the Informix **ESQL** or **4GL** product you are using.

The general use of the INSERT, UPDATE, and DELETE statements is covered in Chapter 4, “Modifying Data.” This chapter examines their use from within a program. It is quite easy to put the statements in a program, but it can be quite difficult to handle errors and to deal with concurrent modifications from multiple programs.

Using DELETE

A program deletes rows from a table by executing a DELETE statement. The DELETE statement can specify rows in the usual way with a WHERE clause, or it can refer to a single row, the last one fetched through a specified cursor.

Whenever you delete rows, you must consider whether rows in other tables depend on the deleted rows. This problem of coordinated deletions is covered in Chapter 4, “Modifying Data.” The problem is the same when deletions are made from within a program.

Direct Deletions

You can embed a DELETE statement in a program. The following example uses **INFORMIX-ESQL/C**:

```
exec sql delete from items
      where order_num = :onum;
```

You can also prepare and execute a statement of the same form dynamically. In either case, the statement works directly on the database to affect one or more rows.

The WHERE clause in the example uses the value of a host variable named **onum**. Following the operation, results are posted in the SQLSTATE and in SQLCA, as usual. The third element of the SQLERRD array contains the count of rows deleted even if an error occurs. The value in SQLCODE shows the overall success of the operation. If it is not negative, no errors occurred and the third element of SQLERRD is the count of all rows that satisfied the WHERE clause and were deleted.

Errors During Direct Deletions

When an error occurs, the statement ends prematurely. The values in SQLSTATE and in SQLCODE and the second element of SQLERRD explain its cause, and the count of rows reveals how many rows were deleted. For many errors, that count is zero because they prevented the database server from beginning the operation at all. For example, if the named table does not exist, or if a column tested in the WHERE clause is renamed, no deletions are attempted.

However, certain errors can be discovered after the operation begins and some rows are processed. The most common of these errors is a lock conflict. The database server must obtain an exclusive lock on a row before it can delete that row. Other programs might be using the rows from the table, preventing the database server from locking a row. Because the issue of locking affects all types of modifications, it is discussed Chapter 7, “Programming for a Multiuser Environment.”

Other, rarer types of error can strike after deletions begin, for example, hardware errors that occur while the database is being updated.

Using Transaction Logging

The best way to prepare for any kind of error during a modification is to use transaction logging. In the event of an error of any kind, you can tell the database server to put the database back the way it was. The following example is based on the preceding **INFORMIX-ESQL/C** example, which is extended to use transactions:

```
exec sql begin work;           /* start the transaction*/
exec sql delete from items
    where order_num = :onum;
del_result = sqlca.sqlcode;    /* save two error */
del_isamno = sqlca.sqlerrd[1]; /* ...code numbers */
del_rowcnt = sqlca.sqlerrd[2]; /* ...and count of rows */
if (del_result < 0)           /* some problem, */
    exec sql rollback work;    /* ...put everything back */
else                           /* everything worked OK, */
    exec sql commit work;      /* ...finish transaction */
```

An important point in this example is that the program saves the important return values in the SQLCA before it ends the transaction. The reason is that both the ROLLBACK WORK and COMMIT WORK statements, as with all SQL statements, set return codes in the SQLCA. Executing a ROLLBACK WORK statement after an error wipes out the error code; unless it was saved, it cannot be reported to the user.

The advantage of using transactions is that no matter what goes wrong, the database is left in a known, predictable state. No question remains about how much of the modification is completed; either all of it is or none of it is.

Coordinated Deletions

The usefulness of transaction logging is particularly clear when you must modify more than one table. For example, consider the problem of deleting an order from the demonstration database. In the simplest form of the problem, you must delete rows from two tables, **orders** and **items**, as shown in the **INFORMIX-4GL** example in Figure 6-1.

```
WHenever Error Continue{do not terminate on error}
Begin Work  {start transaction}
Delete From items
    Where order_num = o_num
If (status >= 0) Then{no error on first delete}
    Delete From orders
        Where order_num = o_num
End If
If (status >= 0) Then{no error on either delete}
    Commit Work
Else      {problem on some delete}
    Display 'Error ', status, ' deleting.'
    Rollback Work
End If
```

Figure 6-1 *A fragment of 4GL that deletes from two tables*

The logic of this program is much the same regardless if transactions are used. If they are not used, the person who sees the error message has a much more difficult set of decisions to make. Depending on when the error occurred one of the following situations applies:

- No deletions were performed; all rows with this order number remain in the database.
- Some, but not all, item rows were deleted; an order record with only some items remains.
- All item rows were deleted, but the order row remains.
- All rows were deleted.

In the second and third cases, the database is corrupted to some extent; it contains partial information that can cause some queries to produce wrong answers. You must take careful action to restore consistency to the information. When transactions are used, all these uncertainties are prevented.

Deleting with a Cursor

You can also write a DELETE statement through a cursor to delete the row that was last fetched. This lets you program deletions based on conditions that cannot be tested in a WHERE clause, as shown in Figure 6-2.

```
int delDupOrder()
{
    int ord_num;
    int dup_cnt, ret_code;

    exec sql declare scan_ord cursor for
        select order_num, order_date
            into :ord_num, :ord_date
            from orders for update;
    exec sql open scan_ord;
    if (sqlca.sqlcode != 0)
        return (sqlca.sqlcode);
    exec sql begin work;
    for(;;)
    {
        exec sql fetch next scan_ord;
        if (sqlca.sqlcode != 0) break;
        dup_cnt = 0; /* default in case of error */
        exec sql select count(*) into dup_cnt from orders
            where order_num = :ord_num;
        if (dup_cnt > 1)
        {
            exec sql delete where current of scan_ord;
            if (sqlca.sqlcode != 0)
                break;
        }
    }
    ret_code = sqlca.sqlcode;
    if (ret_code == 100) /* merely end of data */
        exec sql commit work;
    else /* error on fetch or on delete */
        exec sql rollback work;
    return (ret_code);
}
```

Figure 6-2 **An unsafe ESQL/C function that deletes through a cursor (see Warning)**



Warning: The design of the ESQL/C function in Figure 6-2 is unsafe. It depends for correct operation on the current “isolation level,” which is discussed later in the chapter. Even when it works as intended, its effects depend on the physical order of rows in the table, which is not generally a good idea.

The purpose of the function is to delete rows containing duplicate order numbers. In fact, in the demonstration database, the **orders.order_num** column has a unique index, so duplicate rows cannot occur in it. However, a similar function can be written for another database; this one uses familiar column names.

The function declares **scan_ord**, a cursor to scan all rows in the **orders** table. It is declared with the **FOR UPDATE** clause, which states that the cursor can modify data. If the cursor opens properly, the function begins a transaction and then loops over rows of the table. For each row, it uses an embedded **SELECT** statement to determine how many rows of the table have the order number of the current row. (This step fails without the correct isolation level, as described in Chapter 7, “Programming for a Multiuser Environment.”)

In the demonstration database, with its unique index on this table, the count returned to **dup_cnt** is always 1. However, if it is greater, the function deletes the current row of the table, reducing the count of duplicates by one.

Clean-up functions of this sort are sometimes needed, but they generally need more sophisticated design. This one deletes all duplicate rows except the last one delivered by the database server. That ordering has nothing to do with the contents of the rows or their meanings. You can improve the function in Figure 6-2 by adding, perhaps, an **ORDER BY** clause to the cursor declaration. However, you cannot use **ORDER BY** and **FOR UPDATE** together. A better approach is sketched later in “An Insert Example” on page 6-12.

Using INSERT

You can embed the **INSERT** statement in programs. Its form and use in a program are the same as described in Chapter 4, “Modifying Data,” with the additional feature that you can use host variables in expressions, both in the **VALUES** and **WHERE** clauses. Moreover, a program has the additional ability to insert rows using a cursor.

Using an Insert Cursor

The **DECLARE CURSOR** statement has many variations. Most are used to create cursors for different kinds of scans over data, but one variation creates a special kind of cursor called an *insert cursor*. You use an insert cursor with the **PUT** and **FLUSH** statements to efficiently insert rows into a table in bulk.

Declaring an Insert Cursor

You create an insert cursor by declaring a cursor to be FOR an INSERT statement instead of a SELECT statement. You cannot use such a cursor to fetch rows of data; you can use it only to insert them. An example of the declaration of an insert cursor is shown in Figure 6-3.

```
DEFINE the_company LIKE customer.company,  
      the_fname LIKE customer.fname,  
      the_lname LIKE customer.lname  
DECLARE new_custs CURSOR FOR  
      INSERT INTO customer (company, fname, lname)  
      VALUES (the_company, the_fname, the_lname)
```

Figure 6-3 *A 4GL fragment that declares an insert cursor*

When you open an insert cursor, a buffer is created in memory to hold a block of rows. The buffer receives rows of data as the program produces them; then they are passed to the database server in a block when the buffer is full. This reduces the amount of communication between the program and the database server, and it lets the database server insert the rows with less difficulty. As a result, the insertions go faster.

The minimum size of the insert buffer is set for any implementation of embedded SQL; you have no control over it (it is typically 1 or 2 kilobytes). The buffer is always made large enough to hold at least two rows of inserted values. It is large enough to hold more than two rows when the rows are shorter than the minimum buffer size.

Inserting with a Cursor

The code in Figure 6-3 prepares an insert cursor for use. The continuation, shown in Figure 6-4, shows how the cursor can be used. For simplicity, this example assumes that a function named **next_cust** returns either information about a new customer or null data to signal the end of input.

```
WHENEVER ERROR CONTINUE {do not terminate on error}
BEGIN WORK
OPEN new_custs
WHILE status = 0
    CALL next_cust() RETURNING the_company, the_fname, the_lname
    IF the_company IS NULL THEN
        EXIT WHILE
    END IF
    PUT new_custs
END WHILE
IF status = 0 THEN          {no problem in a PUT}
    FLUSH new_custs        {write any last rows}
END IF
IF status = 0 THEN          {no problem writing}
    COMMIT WORK            {..make it permanent}
ELSE
    ROLLBACK WORK          {retract any changes}
END IF
```

Figure 6-4 *Continuing Figure 6-3, the 4GL code that uses the insert cursor*

The code in Figure 6-4 calls **next_cust** repeatedly. When it returns non-null data, the PUT statement sends the returned data to the row buffer. When the buffer fills, the rows it contains are automatically sent to the database server. The loop normally ends when **next_cust** has no more data to return. Then the FLUSH statement writes any rows that remain in the buffer, after which the transaction terminates.

Examine the INSERT statement in Figure 6-3 once more. The statement by itself, not part of a cursor definition, inserts a single row into the **customer** table. In fact, the whole apparatus of the insert cursor can be dropped from the example code, and the INSERT statement can be written into Figure 6-4 where the PUT statement now stands. The difference is that an insert cursor causes a program to run somewhat faster.

Status Codes After PUT and FLUSH

When a program executes a PUT statement, the program should test whether the row is placed in the buffer successfully. If the new row fits in the buffer, the only action of PUT is to copy the row to the buffer. No errors can occur in this case. However, if the row does not fit, the entire buffer load is passed to the database server for insertion, and an error can occur.

The values returned into the SQL Communications Area give the program the information it needs to sort out each case. SQLCODE is set after every PUT statement—to zero if no error occurs and to a negative error code if an error occurs.

The third element of SQLERRD is set to the number of rows actually inserted into the table: It is set to zero if the new row is merely moved to the buffer; to the count of rows that are in the buffer, if the buffer load is inserted without error; or to the count of rows inserted before an error occurs, if one does occur.

Read the code (Figure 6-4) once again to see how SQLCODE is used. First, if the OPEN statement yields an error, the loop is not executed (because the WHILE condition fails), the FLUSH operation is not performed, and the transaction rolls back.

Second, if the PUT statement returns an error, the loop ends (because of the WHILE condition), the FLUSH operation is not performed, and the transaction rolls back. This can occur only if the loop generates enough rows to fill the buffer at least once; otherwise, the PUT statement cannot generate an error.

The program might end the loop with rows still in the buffer, possibly without inserting any rows. At this point, the SQL status is zero, and the FLUSH operation occurs. If the FLUSH operation produces an error code, the transaction rolls back. Only when all inserts are successfully performed is the transaction committed.

Rows of Constants

The insert cursor mechanism supports one special case where high performance is easy to obtain. This is the case in which all of the values listed in the INSERT statement are constants—no expressions and no host variables, just literal numbers and strings of characters. No matter how many times such an INSERT operation occurs, the rows it produces are identical. In that case, there is no point in copying, buffering, and transmitting each identical row.

Instead, for this kind of INSERT operation, the PUT statement does nothing except to increment a counter. When a FLUSH operation is finally performed, a single copy of the row, and the count of inserts, is passed to the database server. The database server creates and inserts that many rows in one operation.

It is not common to insert a quantity of identical rows. You can do it when you first establish a database, to populate a large table with null data.

An Insert Example

“Deleting with a Cursor” on page 6-7, contains an example of the DELETE statement whose purpose is to look for and delete duplicate rows of a table. A better way to do the same thing is to select the desired rows instead of deleting the undesired ones. The code in Figure 6-5 shows one way to do this. The example is written in **INFORMIX-4GL** to take advantage of some features that make SQL programming easy.

```

BEGIN WORK
INSERT INTO new_orders
  SELECT * FROM ORDERS main
    WHERE 1 = (SELECT COUNT(*) FROM ORDERS minor
              WHERE main.order_num = minor.order_num)
COMMIT WORK

DEFINE ord_row RECORD LIKE orders,
  last_ord LIKE orders.order_num
DECLARE dup_row CURSOR FOR
  SELECT * FROM ORDERS main INTO ord_row.*
    WHERE 1 < (SELECT COUNT(*) FROM ORDERS minor
              WHERE main.order_num = minor.order_num)
    ORDER BY order_date
DECLARE ins_row CURSOR FOR
  INSERT INTO new_orders VALUES (ord_row.*)

BEGIN WORK
OPEN ins_row
LET last_ord = -1
FOREACH dup_row
  IF ord_row.order_num <> last_ord THEN
    PUT ins_row
    LET last_ord = ord_row.order_num
  END IF
END FOREACH
CLOSE ins_row
COMMIT WORK

```

Figure 6-5 *A 4GL program that re-creates a table without duplicates*

This example begins with an ordinary INSERT statement that finds all the nonduplicated rows of the table and inserts them into another table, presumably created before the program started. That leaves only the duplicate rows. (Remember, in the demonstration database the **orders** table has a unique index and cannot have duplicate rows. This example deals with some other database.)

In **INFORMIX-4GL**, you can define a data structure *like* a table; the structure is automatically given one element for each column in the table. The **ord_row** structure is a buffer to hold one row of the table.

The code in Figure 6-5 then declares two cursors. The first, called **dup_row**, returns the duplicate rows in the table. Because it is for input only, it can use the ORDER BY clause to impose some order on the duplicates other than the physical record order used in Figure 6-2 on page 6-7. In this example, the duplicate rows are ordered by their dates (the oldest one remains), but you can use any other order based on the data.

The second cursor is an insert cursor. It is written to take advantage of the *asterisk* notation of **INFORMIX-4GL**; you can supply values for all columns simply by naming a record with an asterisk to indicate *all fields*.

The remainder of the code examines the rows returned through **dup_row**. It inserts the first one from each group of duplicates into the new table and disregards the rest.

This example uses the simplest kind of error handling. Unless told otherwise, an **INFORMIX-4GL** program automatically terminates when an error code is set in **SQLCODE**. In this event, the active transaction rolls back. This program relies on that behavior; it assumes that if it reaches the end, no errors exist and the transaction can be committed. This kind of error handling is acceptable when errors are unlikely, and the program is used by people who do not need to know why the program terminates.

Using UPDATE

You can embed the **UPDATE** statement in a program in any of the forms described in Chapter 4, “Modifying Data,” with the additional feature that you can name host variables in expressions, both in the **SET** and **WHERE** clauses. Moreover, a program can update the row addressed by a cursor.

How Many Rows Were Affected? **SQLCODE** and **SQLERRD**

When your program uses a cursor to select rows, it can test **SQLCODE** for 100, the end-of-data return code. This is set to indicate that no rows, or no more rows, satisfy the query conditions. The end-of-data return code is set in **SQLCODE** only following **SELECT** statements; it is not used following **DELETE**, **INSERT**, or **UPDATE** statements.

A query that finds no data is not a success. However, an **UPDATE** or **DELETE** statement that happens to update or delete no rows is still considered a success: it updated or deleted the set of rows that its **WHERE** clause said it should; however, the set was empty.

In the same way, the **INSERT** statement does not set the end-of-data return code even when the source of the inserted rows is a **SELECT** statement and it selected no rows. The **INSERT** statement is a success because it inserted as many rows as it was asked to do (that is, zero).

To find out how many rows are inserted, updated, or deleted, a program can test the third element of **SQLERRD**. The count of rows is there, regardless of the value (zero or negative) in **SQLCODE**.

Using an Update Cursor

An *update cursor* permits you to delete or update the current row; that is, the most recently fetched row. The following example (in **INFORMIX-ESQL/COBOL**) show the declaration of an update cursor:

```
EXEC SQL
    DECLARE names CURSOR FOR
        SELECT fname, lname, company
        FROM customer
        FOR UPDATE
END-EXEC
```

The program that uses this cursor can fetch rows in the usual way.

```
EXEC SQL
    FETCH names INTO :FNAME, :LNAME, :COMPANY
END-EXEC.
```

If the program then decides that the row needs to be changed, it can do so.

```
IF COMPANY IS EQUAL TO 'SONY'
    EXEC SQL
        UPDATE customer
            SET fname = 'Midori', lname = 'Tokugawa'
            WHERE CURRENT OF names
    END-EXEC.
```

The words `CURRENT OF names` take the place of the usual test expressions in the `WHERE` clause. In other respects, the `UPDATE` statement is the same as usual—even including the specification of the table name, which is implicit in the cursor name but still required.

The Purpose of the Keyword UPDATE

The purpose of the keyword `UPDATE` in a cursor is to let the database server know that the program can update (or delete) any row that it fetches. The database server places a more demanding lock on rows that are fetched through an update cursor and a less demanding lock when fetching a row for a cursor that is not declared with that keyword. This results in better performance for ordinary cursors and a higher level of concurrent use in a multiprocessing system. (Levels of locks and concurrent use are discussed in Chapter 7, “Programming for a Multiuser Environment.”)

Updating Specific Columns

You can declare a cursor to update specific columns. Here is a revision of the preceding **ESQL/COBOL** example has been modified to use this feature and is shown in the following example:

```
EXEC SQL
    DECLARE names CURSOR FOR
        SELECT fname, lname, company, phone
            INTO :FNAME,:LNAME,:COMPANY,:PHONE FROM customer
    FOR UPDATE OF fname, lname
END-EXEC.
```

Only the **fname** and **lname** columns can be updated through this cursor. A statement such as the following one is rejected as an error:

```
EXEC SQL
    UPDATE customer
        SET company = 'Siemens'
        WHERE CURRENT OF names
END-EXEC.
```

If the program attempts such an update, an error code is returned and no update takes place. An attempt to delete using **WHERE CURRENT OF** is also rejected, because deletion affects all columns.

UPDATE Keyword Not Always Needed

The ANSI standard for SQL does not provide for the **FOR UPDATE** clause in a cursor definition. When a program uses an ANSI-compliant database, it can update or delete using any cursor.

Cleaning up a Table

A final, hypothetical example of using an update cursor presents a problem that should never arise with an established database but could arise in the initial design phases of an application.

A large table, named **target**, for example, is created and populated. A character column, **datcol**, inadvertently acquires some null values. These rows should be deleted. Furthermore, a new column, **serials**, is added to the

table with the ALTER TABLE statement. This column is to have unique integer values installed. Figure 6-6 shows the INFORMIX-ESQL/C code needed to accomplish these things.

```
exec sql begin declare section;
char dcol[80];
short dcolint;
int sequence;
exec sql end declare section;

exec sql declare target_row cursor for
select datcol
into :dcol:dcolint
from target
for update of serials;
exec sql begin work;
exec sql open target_row;
if (sqlca.sqlcode == 0) exec sql fetch next target_row;
for(sequence = 1; sqlca.sqlcode == 0; ++sequence)
{
    if (dcolint < 0) /* null datcol */
        exec sql delete where current of target_row;
    else
        exec sql update target set serials = :sequence
        where current of target_row;
}
if (sqlca.sqlcode >= 0) exec sql commit work;
else exec sql rollback work;
```

Figure 6-6 *Cleaning up a table using an update cursor*

Summary

A program can execute the INSERT, DELETE, and UPDATE statements as described in Chapter 4, “Modifying Data.” A program also can scan through a table with a cursor, updating or deleting selected rows. It also can use a cursor to insert rows, with the benefit that the rows are buffered and sent to the database server in blocks.

In all these activities, the program must take pains to detect errors and return the database to a known state when one occurs. The most important tool for doing this is the transaction. Without transaction logging, the program has a more difficult time recovering from errors.

Summary

Programming for a Multiuser Environment

Chapter Overview	3
Concurrency and Performance	3
Locking and Integrity	3
Locking and Performance	3
Concurrency Issues	4
How Locks Work	5
Kinds of Locks	6
Lock Scope	6
Database Locks	6
Table Locks	7
Page, Row, and Key Locks	8
The Duration of a Lock	8
Locks While Modifying	9
Setting the Isolation Level	9
Dirty Read Isolation	10
Committed Read Isolation	10
Cursor Stability Isolation	11
Repeatable Read Isolation	12

Setting the Lock Mode	13
Waiting for Locks	13
Not Waiting for Locks	13
Waiting a Limited Time	14
Handling a Deadlock	14
Handling External Deadlock	14
Simple Concurrency	15
Locking with Other Database Servers	15
Isolation While Reading	16
Locking Updated Rows	17
Hold Cursors	17
Summary	19

Chapter Overview

If your database is contained in a single-user workstation and is not connected on a network to other machines, your programs can modify data freely. But in all other cases, you must allow for the possibility that, while your program is modifying data, another program is reading or modifying the same data. This is *concurrency*: two or more independent uses of the same data at the same time. This chapter addresses concurrency, locking, and isolation levels.

Concurrency and Performance

Concurrency is crucial to good performance in a multiprogramming system. When access to the data is *serialized* so that only one program at a time can use it, processing slows dramatically.

Locking and Integrity

Unless controls are placed on the use of data, concurrency can lead to a variety of negative effects. Programs can read obsolete data or modifications can be lost even though they were apparently completed.

The database server prevents errors of this kind by imposing a system of *locks*. A lock is a claim, or reservation, that a program can place on a piece of data. The database server guarantees that, as long as the data is locked, no other program can modify it. When another program requests the data, the database server either makes the program wait or turns it back with an error.

Locking and Performance

Because a lock serializes access to one piece of data, it reduces concurrency; any other programs that want access to that data must wait. The database server can place a lock on a single row, a disk page (which holds multiple

rows), a whole table, or an entire database. The more locks it places and the larger the objects it locks, the more concurrency is reduced. The fewer the locks and the smaller the objects, the greater concurrency and performance can be.

This section discusses how a program can achieve two goals:

- To place all the locks needed to ensure data integrity
- To lock the fewest, smallest pieces of data possible consistent with the preceding goal

Concurrency Issues

To understand the hazards of concurrency, you must think in terms of multiple programs, each executing at its own speed. The following example suppose that your program is fetching rows through the following cursor:

```
DECLARE sto_course CURSOR FOR
  SELECT * FROM stock
    WHERE manu_code = 'ANZ'
```

The transfer of each row from the database server to the program takes time. During and between transfers, other programs can perform other database operations. Say that at about the same time your program fetches the rows produced by that query, another user's program executes the following update:

```
UPDATE stock
  SET unit_price = 1.15 * unit_price
  WHERE manu_code = 'ANZ'
```

In other words, both programs are reading through the same table, one fetching certain rows and the other changing the same rows. The following possibilities concerning what happens next:

1. The other program finishes its update before your program fetches its first row.
Your program shows you only updated rows.
2. Your program fetches every row before the other program has a chance to update it.
Your program shows you only original rows.

3. After your program fetches some original rows, the other program catches up and goes on to update some rows that your program has yet to read; then it executes the COMMIT WORK statement.

Your program might return a mixture of original rows and updated rows.

4. Same as number 3, except that after updating the table, the other program issues a ROLLBACK WORK statement.

Your program can show you a mixture of original rows and updated rows that no longer exist in the database.

The first two possibilities are harmless. In number 1, the update is complete before your query begins. It makes no difference whether it finished a microsecond or a week ago.

In number 2, your query is, in effect, complete before the update begins. The other program might have been working just one row behind yours, or it might not start until tomorrow night; it does not matter.

The second pair of chances, however, can be very important to the design of some applications. In number 3, the query returns a mix of updated and original data. That can be a negative thing in some applications. In others, for instance one that is taking an average of all prices, it might not matter at all.

In number 4, it can be disastrous if a program returns some rows of data that, because their transaction was cancelled, can no longer be found in the table.

Another concern arises when your program uses a cursor to update or delete the last-fetched row. Erroneous results occur with the following sequence of events:

- Your program fetches the row.
- Another program updates or deletes the row.
- Your program updates or deletes WHERE CURRENT OF *names*.

You control concurrent events such as these using the locking and *isolation level* features of the database server.

How Locks Work

The **INFORMIX-OnLine Dynamic Server** database server supports a complex, flexible set of locking features that is described in this section. Some of these locking features work differently on an **INFORMIX-SE** database server. (See Chapter 1 of the *Informix Guide to SQL: Reference* for a summary of the differences between locking in **SE** and **OnLine** database servers.)

Kinds of Locks

INFORMIX-OnLine Dynamic Server supports the following three kinds of locks, which it uses in different situations:

<i>shared</i>	A shared lock reserves its object for reading only. It prevents the object from changing while the lock remains. More than one program can place a shared lock on the same object.
<i>exclusive</i>	<p>An exclusive lock reserves its object for the use of a single program. It is used when the program intends to change the object.</p> <p>An exclusive lock cannot be placed where any other kind of lock exists. Once one has been placed, no other lock can be placed on the same object.</p>
<i>promotable</i>	A promotable lock establishes the intent to update. It can only be placed where no other promotable or exclusive lock exists. Promotable locks can exist with shared locks. They are “promoted” to an exclusive lock.

Lock Scope

You can apply locks to entire databases, entire tables, disk pages, single rows, or index-key values. The size of the object being locked is referred to as the *scope* of the lock (also called the *lock granularity*). In general, the larger the scope of a lock, the more concurrency is reduced but the simpler programming becomes.

Database Locks

You can lock an entire database. The act of opening a database places a shared lock on the name of the database. A database is opened with the CONNECT, DATABASE, or CREATE DATABASE statements. As long as a program has a database open, the shared lock on the name prevents any other program from dropping the database or putting an exclusive lock on it.

You can lock an entire database exclusively with the following statement:

```
DATABASE database name EXCLUSIVE
```

This statement succeeds if no other program has opened that database. Once the lock is placed, no other program can open the database, even for reading because its attempt to place a shared lock on the database name fails.

A database lock is released only when the database closes. That can be done explicitly with the DISCONNECT or CLOSE DATABASE statements or implicitly by executing another DATABASE statement.

Because locking a database reduces concurrency in that database to zero, it makes programming very simple—concurrent effects cannot happen. However, you should lock a database only when no other programs need access. Database locking is often used before applying massive changes to data during off-peak hours.

Table Locks

You can lock entire tables. In some cases, this is done automatically. **INFORMIX-OnLine Dynamic Server** always locks an entire table while it performs any of the following statements:

- ALTER INDEX
- ALTER TABLE
- CREATE INDEX
- DROP INDEX
- RENAME COLUMN
- RENAME TABLE

The completion of the statement (or end of the transaction) releases the lock. An entire table also can be locked automatically during certain queries.

You can use the LOCK TABLE statement to lock an entire table explicitly. This statement allows you to place either a shared lock or an exclusive lock on an entire table.

A shared table lock prevents any concurrent updating of that table while your program is reading from it. **INFORMIX-OnLine Dynamic Server** achieves the same degree of protection by setting the isolation level, as described in the next section; this allows greater concurrency. However, all Informix database servers support the LOCK TABLE statement.

An exclusive table lock prevents any concurrent use of the table. This has a serious effect on performance if many other programs are contending for the use of the table. As with an exclusive database lock, an exclusive table lock is often used when massive updates are applied during off-peak hours. For example, some applications do not update tables during the hours of peak use. Instead, they write updates to an *update journal*. During off-peak hours, that journal is read and all updates are applied in a batch.

Page, Row, and Key Locks

One row of a table is the smallest object that can be locked. A program can lock one row or a selection of rows while other programs continue to work on other rows of the same table.

INFORMIX-OnLine Dynamic Server stores data in units called *disk pages*. (Its disk-storage methods are described in detail in Chapter 10, “Tuning Your Data Model.”) A disk page contains one or more rows. In some cases, it is better to lock a disk page than to lock individual rows on it.

You choose between locking by rows or locking by pages when you create the table. **INFORMIX-OnLine Dynamic Server** supports a clause, `LOCK MODE`, to specify either page or row locking. You can specify lock mode in the `CREATE TABLE` statement and later change it with the `ALTER TABLE` statement. (Other Informix database servers do not offer the choice; they lock by row or by page, whichever makes the better implementation.)

Page and row locking are used identically. Whenever **INFORMIX-OnLine Dynamic Server** needs to lock a row, it locks either the row itself or the page it is on, depending on the lock mode established for the table.

In certain cases, the database server has to lock a row that does not exist—in effect, it locks the place in the table where the row would be if it did exist. The database server does this by placing a lock on an index-key value. Key locks are used identically to row locks. When the table uses row locking, key locks are implemented as locks on imaginary rows. When it uses page locking, a key lock is placed on the index page that contains the key or that would contain the key if the it existed.

The Duration of a Lock

The program controls the duration of a database lock. A database lock is released when the database closes.

The duration of a table lock depends on whether the database uses transactions. If it does not (that is, if no transaction log exists and the `COMMIT WORK` statement is not used), a table lock remains until it is removed by the execution of the `UNLOCK TABLE` statement.

The duration of table, row, and index locks depends on what SQL statements are used and on whether transactions are in use.

When transactions are used, the end of a transaction releases all table, row, page, and index locks. This is an important point: When a transaction ends, *all locks are released*.

Locks While Modifying

When the database server fetches a row through an update cursor, it places a promotable lock on the fetched row. If this succeeds, the database server knows that no other program can alter that row. Because a promotable lock is not exclusive, other programs can continue to read the row. This helps performance because the program that fetched the row can take some time before it issues the UPDATE or DELETE statement, or it can simply fetch the next row.

When it is time to modify a row, the database server obtains an exclusive lock on the row. If it already had a promotable lock, it changes it to exclusive status.

The duration of an exclusive row lock depends on whether transactions are in use. If they are not in use, the lock is released as soon as the modified row is written to disk. When transactions are in use, all such locks are held until the end of the transaction. This prevents other programs from using rows that might be rolled back to their original state.

When transactions are in use, a key lock is used whenever a row is deleted. This prevents the following error from occurring:

- Program A deletes a row.
- Program B inserts a row having the same key.
- Program A rolls back its transaction, forcing the database server to restore its deleted row. What is to be done with the row inserted by program B?

By locking the index, the database server prevents a second program from inserting a row until the first program commits its transaction.

The locks placed while reading a row are controlled by the current isolation level, which is discussed in the next section.

Setting the Isolation Level

The *isolation level* is the degree to which your program is isolated from the concurrent actions of other programs. **INFORMIX-OnLine Dynamic Server** offers a choice of four isolation levels. It implements them by setting different rules for how a program uses locks when it is reading. (This does not apply to reads performed on update cursors.)

You set the isolation level using the command SET ISOLATION LEVEL. All **INFORMIX-OnLine Dynamic Server** database servers support this statement.

Dirty Read Isolation

The simplest isolation level, Dirty Read, amounts to virtually no isolation. When a program fetches a row, it places no locks and it respects none; it simply copies rows from the database without regard for what other programs are doing.

A program always receives complete rows of data; even under Dirty Read isolation, a program never sees a row in which some columns have been updated and some have not. However, a program using Dirty Read isolation sometimes reads updated rows before the updating program ends its transaction. If the updating program later rolls back its transaction, the reading program processed data that never really existed (number 4 in the list of concurrency issues on page 7-5).

Dirty Read is the most efficient level. The reading program never waits and never makes another program wait. It is the preferred level in any of the following cases:

- All tables are static; that is, concurrent programs only read and never modify data.
- The database is held in an exclusive lock.
- It is certain that only one program is using the database.

Committed Read Isolation

When a program requests the Committed Read isolation level, **INFORMIX-OnLine Dynamic Server** guarantees that it never returns a row that is not committed to the database. This prevents the situation in number 4 in the list of concurrency issues on page 7-5 (reading data that is not committed and that is subsequently rolled back).

Committed Read is implemented very simply. Before fetching a row, the database server tests to determine whether an updating process placed a lock on the row. If not, it returns the row. Because rows that are updated but not committed have locks on them, this test ensures that the program does not read uncommitted data.

Committed Read does not actually place a lock on the fetched row; hence, it is almost as efficient as Dirty Read. It is appropriate for use when each row of data is processed as an independent unit, without reference to other rows in the same or other tables.

Cursor Stability Isolation

The next level is called Cursor Stability. When it is in effect, the database server places a lock on the latest row fetched. It places a shared lock for an ordinary cursor or a promotable lock for an update cursor. Only one row is locked at a time; that is, each time a row is fetched, the lock on the previous row is released (unless that row is updated, in which case the lock holds until the end of the transaction).

Cursor Stability ensures that a row does not change while the program examines it. This is important when the program updates some other table based on the data it reads from this row. Because of Cursor Stability, the program is assured that the update is based on current information. It prevents the use of *stale data*.

Here is an example to illustrate this point. In terms of the demonstration database, Program A wants to insert a new stock item for manufacturer Hero. Concurrently, program B wants to delete manufacturer Hero and all stock associated with it. The following sequence of events can occur:

1. Program A, operating under Cursor Stability, fetches the Hero row from the **manufact** table to learn the manufacturer code: This places a shared lock on the row.
2. Program B issues a DELETE statement for that row. Because of the lock, the database server makes the program wait.
3. Program A inserts a new row in the **stock** table using the manufacturer code it obtained from the **manufact** table.
4. Program A closes its cursor on the **manufact** table or reads a different row of it, releasing its lock.
5. Program B, released from its wait, completes the deletion of the row and goes on to delete the rows of **stock** that use manufacturer code HRO, including the row just inserted by Program A.

If Program A used a lesser level of isolation, the following sequence could have happened:

1. Program A reads the Hero row of the **manufact** table to learn the manufacturer code. No lock is placed.
2. Program B issues a DELETE statement for that row. It succeeds.
3. Program B deletes all rows of **stock** that use manufacturer code HRO.

4. Program *B* ends.
5. Program *A*, not aware that its copy of the Hero row is now invalid, inserts a new row of **stock** using the manufacturer code HRO.
6. Program *A* ends.

At the end, a row occurs in **stock** that has no matching manufacturer code in **manufact**. Furthermore, Program *B* apparently has a bug; it did not delete the rows it was supposed to delete. The use of the Cursor Stability isolation level prevents these effects.

(It is possible to rearrange the preceding scenario so that it fails even with Cursor Stability. All that is required is for Program *B* to operate on tables in the reverse sequence to Program *A*. If Program *B* deletes from **stock** before it removes the row of **manufact**, no degree of isolation can prevent an error. Whenever this kind of error is possible, it is essential that all programs involved use the same sequence of access.)

Because Cursor Stability locks only one row at a time, it restricts concurrency less than does a table lock or database lock.

Repeatable Read Isolation

The Repeatable Read isolation level asks the database server to put a lock on every row the program examines and fetches. The locks that are placed are shareable for an ordinary cursor and promotable for an update cursor. The locks are placed individually as each row is examined. They are not released until the cursor closes or a transaction ends.

Repeatable Read allows a program that uses a scroll cursor to read selected rows more than once and to be sure that they are not modified or deleted between readings. (Scroll cursors are described in Chapter 5, “Programming with SQL.”) No lower isolation level guarantees that rows still exist and are unchanged the second time they are read.

Repeatable Read isolation places the largest number of locks and holds them the longest. Therefore, it is the level that reduces concurrency the most. If your program uses this level of isolation, you must think carefully about how many locks it places, how long they are held, and what the effect can be on other programs.

In addition to the effect on concurrency, the large number of locks can be a problem. The database server records only a fixed number of locks for all the programs it serves. (This number is adjustable.) If that table fills up, the

database server cannot place a lock and returns an error code. The person who administers an **INFORMIX-OnLine Dynamic Server** system can monitor the lock table and tell you when it is heavily used.

Repeatable Read is automatically the isolation level in an ANSI-compliant database. Repeatable Read is required to ensure operations behave in accordance with the ANSI standard for SQL.

Setting the Lock Mode

The lock mode determines what happens when your program encounters locked data. One of the following things occurs when a program attempts to fetch or modify a locked row:

- The program receives an immediate return from the database server with an error code in SQLCODE.
- The program is suspended until the lock is removed by the program that placed it.
- The program is suspended for a time and then, if the lock is not removed, it receives an error-return code from the database server.

You choose among these results with the command SET LOCK MODE.

Waiting for Locks

If you prefer to wait (and this is the best choice for many applications), execute the following command:

```
SET LOCK MODE TO WAIT
```

When this lock mode is set, your program usually ignores the existence of other concurrent programs. When it needs to access a row that another program has locked, your program waits until the lock is removed, then proceeds. The delays are usually imperceptible.

Not Waiting for Locks

The disadvantage of waiting for locks is that the wait might become very long (although properly designed applications should hold their locks very briefly). When the possibility of a long delay is not acceptable, a program can execute the following command:

```
SET LOCK MODE TO NOT WAIT
```

When the program requests a locked row, it immediately receives an error code (for example, error -107, *Record is locked*), and the current SQL statement terminates. It is up to the program to roll back its current transaction and try again.

The initial setting is *not waiting* when a program starts up. If you are using SQL interactively and see an error related to locking, set the lock mode to wait. If you are writing a program, consider making that one of the first embedded SQL commands that the program executes.

Waiting a Limited Time

When you use **INFORMIX-OnLine Dynamic Server**, you have an additional choice: You can ask the database server to set an upper limit on a wait. You can issue the following command:

```
SET LOCK MODE TO WAIT 17
```

This places an upper limit of 17 seconds on the length of any wait. If a lock is not removed in that time, the error code is returned.

Handling a Deadlock

A *deadlock* is a situation in which a pair of programs block the progress of each other. Each program has a lock on some object that the other program wants to access. A deadlock arises only when all programs concerned set their lock modes to wait for locks.

INFORMIX-OnLine Dynamic Server detects deadlocks immediately when they involve only data at a single network server. It prevents the deadlock from occurring by returning an error code (error -143 *ISAM error: deadlock detected*) to the second program to request a lock. The error code is

the one the program receives if it sets its lock mode to not wait for locks. Thus, if your program receives an error code related to locks even after it sets lock mode to wait, you know the cause is an impending deadlock.

Handling External Deadlock

A deadlock can also occur between programs on different database servers. In this case, **INFORMIX-OnLine Dynamic Server** cannot instantly detect the deadlock. (Perfect deadlock detection requires excessive communications traffic among all database servers in a network.) Instead, each database server sets an upper limit on the amount of time that a program can wait to obtain a lock on data at a different database server. If the time expires, the database server assumes that a deadlock was the cause and returns a lock-related error code.

In other words, when external databases are involved, every program runs with a maximum lock-waiting time. The maximum is set for the database server and can be changed by the database administrator.

Simple Concurrency

If you are not sure which choice to make concerning locking and concurrency, and if your application is straightforward, have your program execute the following commands when it starts up (immediately after the first DATABASE statement):

```
SET LOCK MODE TO WAIT
SET ISOLATION TO REPEATABLE READ
```

Ignore the return codes from both statements. Proceed as if no other programs exist. If no performance problems arise, you do not need to read this section again.

Locking with Other Database Servers

INFORMIX-OnLine Dynamic Server manages its own locking so that it can provide the different kinds of locks and levels of isolation described in the preceding topics. Other Informix database servers implement locks using the facilities of the host operating system and cannot provide the same conveniences.

Some host operating systems provide locking functions as operating system services. In these systems, database servers support the SET LOCK MODE statement.

Some host operating systems do not provide *kernel-locking* facilities. In these systems, the database server performs its own locking based on small files that it creates in the database directory. These files have the suffix **.lok**.

You can tell the kind of system in which your database server is running by executing the SET LOCK MODE statement and testing the error code, as shown in the following fragment of **INFORMIX-ESQL/C** code:

```
#define LOCK_ONLINE 1
#define LOCK_KERNEL 2
#define LOCK_FILES 3
int which_locks()
{
    int locktype;

    locktype = LOCK_FILES;
    exec sql set lock mode to wait 30;
    if (sqlca.sqlcode == 0)
        locktype = LOCK_ONLINE;
    else
    {
        exec sql set lock mode to wait;
        if (sqlca.sqlcode == 0)
            locktype = LOCK_KERNEL;
    }
    /* restore default condition */
    exec sql set lock mode to not wait;
    return(locktype);
}
```

If the database server does not support the SET LOCK MODE statement, your program is effectively always in NOT WAIT mode; that is, whenever it tries to lock a row that is locked by another program, it receives an error code immediately.

Isolation While Reading

Informix database servers other than **INFORMIX-OnLine Dynamic Server** do not normally place locks when fetching rows. Nothing comparable exists to the shared locks used by **OnLine** to implement the Cursor Stability isolation level.

If your program fetches a row with a singleton **SELECT** statement or through a cursor that is not declared **FOR UPDATE**, the row is fetched immediately, regardless of whether it is locked or modified by an unfinished transaction.

This design produces the best performance, especially when locks are implemented by writing notes in disk files, but you must be aware that the program can read rows that are modified by uncommitted transactions.

You can obtain the effect of Cursor Stability isolation by declaring a cursor **FOR UPDATE**, and then using it for input. Whenever the database server fetches a row through an update cursor, it places a lock on the fetched row. (If the row is already locked, the program waits or receives an error, depending on the lock mode.) When the program fetches another row without updating the current one, the lock on the current row is released and the new row is locked.

You can be sure by fetching through an update cursor, that the fetched row is locked as long as you use it. (The row cannot become *stale*.) You are also assured of fetching only committed data, because locks on rows that are updated are held until the end of the transaction. Depending on the host operating system and the database server, you might experience a performance penalty for using an update cursor this way.

Locking Updated Rows

When a cursor is declared **FOR UPDATE**, locks are handled as follows: Before a row is fetched, it is locked; or if it cannot be locked, the program waits or returns an error.

The next time a fetch is requested, the database server notes whether the current row is modified (using either the **UPDATE** or **DELETE** statement with **WHERE CURRENT OF**) and whether a transaction is in progress. If both these things are true, the lock on the row is retained. Otherwise, the lock is released.

So if you perform updates within a transaction, all updated rows remain locked until the transaction ends. Rows that are not updated are locked only while they are current. Rows updated outside a transaction, or in a database that does not use transaction logging, are also unlocked as soon as another row is fetched.

Hold Cursors

When transaction logging is used, the database server guarantees that anything done within a transaction can be rolled back at the end of it. To do this reliably, the database server normally applies the following rules:

- All cursors are closed by ending a transaction.
- All locks are released by ending a transaction.

These rules are normal with all database systems that support transactions, and for most applications they do not cause any trouble. However, circumstances exist where using standard transactions with cursors are not possible. For example, a situation is sketched in Figure 7-1 which works fine without transactions. However, when transactions are added, closing the cursor conflicts with using two cursors simultaneously.

```
DECLARE master CURSOR FOR ...
DECLARE detail CURSOR FOR ... FOR UPDATE
OPEN master
LOOP:
  FETCH master INTO ...
  IF (the fetched data is appropriate) THEN
    BEGIN WORK
    OPEN detail USING data read from master
    FETCH detail ...
    UPDATE ... WHERE CURRENT OF detail
    COMMIT WORK
  END IF
END LOOP
CLOSE MASTER
```

Figure 7-1 *A pseudocode sketch of one common form of database application*

In this design, one cursor is used to scan a table. Selected records are used as the basis for updating a different table. The problem is that when each update is treated as a separate transaction (as shown in the pseudocode in Figure 7-1), the COMMIT WORK statement following the UPDATE closes all cursors—including the master cursor.

The simplest alternative is to move the COMMIT WORK and BEGIN WORK statements to be the last and first ones, respectively, so that the entire scan over the master table is one large transaction. This is sometimes possible, but it may become impractical if many rows need to be updated. The number of locks can be too large, and they are held for the duration of the program.

A solution supported by Informix database servers is to add the keywords `WITH HOLD` to the declaration of the master cursor. Such a cursor is referred to as a *hold cursor* and is not closed at the end of a transaction. The database server still closes all other cursors, and it still releases all locks, but the hold cursor remains open until it is explicitly closed.

Before you attempt to use a hold cursor, you must be sure that you understand the locking mechanism described here, and you must also understand the programs that are running concurrently. The reason is that whenever `COMMIT WORK` is executed, all locks are released, including any locks placed on rows fetched through the hold cursor.

This has little importance if the cursor is used as intended, for a single forward scan over a table. However, you are allowed to specify `WITH HOLD` for any cursor, including update cursors and scroll cursors. Before you do this, you must understand the implications of the fact that all locks (including locks on entire tables) are released at the end of a transaction.

Summary

Whenever multiple programs have access to a database concurrently (and when at least one of them can modify data), all programs must allow for the possibility that another program can change the data even as they read it. The database server provides a mechanism of locks and isolation levels that usually allow programs to run as if they were alone with the data.

Summary



***Section II:* Designing and Managing Databases**





Building Your Data Model

Chapter Overview	3
Why Build a Data Model	3
Entity-Relationship Data Model Overview	3
Identifying and Defining Your Principal Data Objects	4
Discovering Entities	4
Choosing Possible Entities	5
Pruning Your List of Entities	5
The Telephone-Directory Example	6
Diagramming Your Entities	8
Defining the Relationships	8
Connectivity	8
Existence Dependency	9
Cardinality	9
Discover the Relationships	10
Diagramming Your Relationships	14
Identifying Attributes	14
How to Identify an Attribute	15
Selecting Attributes for Your Entities	15
Listing Your Attributes	16
About Entity Occurrences	16
Diagramming Your Data Objects	16
How You Read Entity-Relationship Diagrams	18
The Telephone-Directory Example	18
What's Next	19

Translating E-R Data Objects Into Relational Constructs	20
Rules for Defining Tables, Rows, and Columns	20
Placing Constraints on Columns	21
Determining Keys for Tables	22
Primary Keys	22
Foreign Keys (Join Columns)	24
Adding Keys to the Telephone-Directory Diagram	25
Resolving Your Relationships	26
Resolving m:n Relationships	26
Resolving Other Special Relationships	28
Normalizing Your Data Model	29
First Normal Form	29
Second Normal Form	31
Third Normal Form	32
Summary of Normalization Rules	32
Summary	33

Chapter Overview

The first step in creating a database is to construct a data model—a precise, complete definition of the data to be stored. This chapter contains a cursory overview of one method of doing this. The following chapters describe how to implement a data model once you design it.

To understand the material in this chapter, you should have a basic understanding of SQL and relational database theory.

Why Build a Data Model

You may already have an idea of a database. If you use some type of formal notation to build your data model, you can help your design in two ways:

- It makes you think through the data model completely.
A mental model often contains unexamined assumptions; formalizing the design reveals these points.
- It is easier to communicate your design to other people.
A formal statement makes the model explicit, so that others can return comments and suggestions in the same form.

Entity-Relationship Data Model Overview

Different books present different formal methods of modeling data. Most methods force you to be thorough and precise. If you have already learned some method, by all means use it.

This chapter presents a summary of the entity-relationship (E-R) data model, a modeling method taught in training courses presented by Informix and other companies and based on the original data model by E. F. Codd. This modeling method is carried out in the following steps:

1. Identify and define the principal data objects (entities, relationships, and attributes).
2. Diagram the data objects using the entity-relationship approach.
3. Translate your entity-relationship data objects into relational constructs.
4. Resolve the logical data model.
5. Normalize the logical data model.

Steps 1-5 are discussed in this chapter. Chapter 9, “Implementing Your Data Model,” discusses a final step—converting your logical data model to a physical schema.

The end product of data modeling is a fully defined database design encoded in a diagram like that in Figure 8-16 on page 8-31, which shows the final set of tables for a personal telephone directory. The personal telephone directory is an example developed in this chapter. It is used rather than the **stores6** database used in the rest of this book because it is small enough to be developed completely in one chapter but large enough to show the entire method.

Identifying and Defining Your Principal Data Objects

The first step in building an entity-relationship data model is to identify and define your principal data objects. The principal data objects are entities, relationships, and attributes.

Discovering Entities

An *entity* is a principal data object that is of significant interest to the user. It is usually a person, place, thing, or event to be recorded in the database. If the data model were a language, entities would be its nouns. The **stores6** database contains the following entities: *customer*, *orders*, *items*, *stock*, *catalog*, *cust_calls*, *call_type*, *manufact*, and *state*.

The first step in modeling is to choose the entities to record. Most of the entities that you choose will become tables in the model.

Choosing Possible Entities

If you have an idea for your database, you can probably list several entities immediately. However, if other people use the database, you should poll them for their understanding of what fundamental *things* the database should contain. Make a preliminary list of all the entities you can identify. Interview the potential users of the database for their opinions about what must be recorded in the database. Determine basic characteristics for each entity, such as “there must be at least one address associated with a name.” All of the decisions you make in determining your entities become your *business rules*. The section, “The Telephone-Directory Example” on page 8-6, provides some of the business rules for the example shown in this chapter.

Later, when you *normalize* your data model, some of the entities may expand or become other data objects. See “Normalizing Your Data Model” on page 8-29 for additional information.

Pruning Your List of Entities

When the list of entities seems complete, prune it by making sure that each one has the following qualities:

- It is significant.

List only entities that are important to the users of the database and worth the trouble and expense of computer tabulation.

- It is generic.

List only types of things, not individual instances. For instance, *symphony* might be an entity, but *Beethoven's Fifth* would be an entity instance or entity occurrence.

- It is fundamental.

List only entities that exist independently, without needing something else to explain them. Anything you could call a trait, a feature, or a description is not an entity. For example, a *part number* is a feature of the fundamental entity called *part*. Also, do not list things that you can derive from other entities; for example, avoid any sum, average, or other quantity that you can calculate in a SELECT expression.

- It is unitary.

Be sure that each entity you name represents a single class—that it cannot be broken down into subcategories, each with its own features. In planning the telephone-directory model (see “The Telephone-Directory Example” on page 8-6) an apparently simple entity, the telephone number, turns out to consist of three categories, each with different features.

These choices are neither simple nor automatic. To discover the best choice of entities, you must think deeply about the nature of the data you want to store. Of course, that is exactly the point of making a formal data model. The following section describes the example shown in this chapter in further detail.

The Telephone-Directory Example

Suppose that you create a database that computerizes a personal telephone directory. The database model must record the names, addresses, and telephone numbers of people and organizations that its user deals with for business and pleasure.

The first step is to define the entities, and the first thing you might do is look carefully at a page from a telephone directory to see what entities are there.

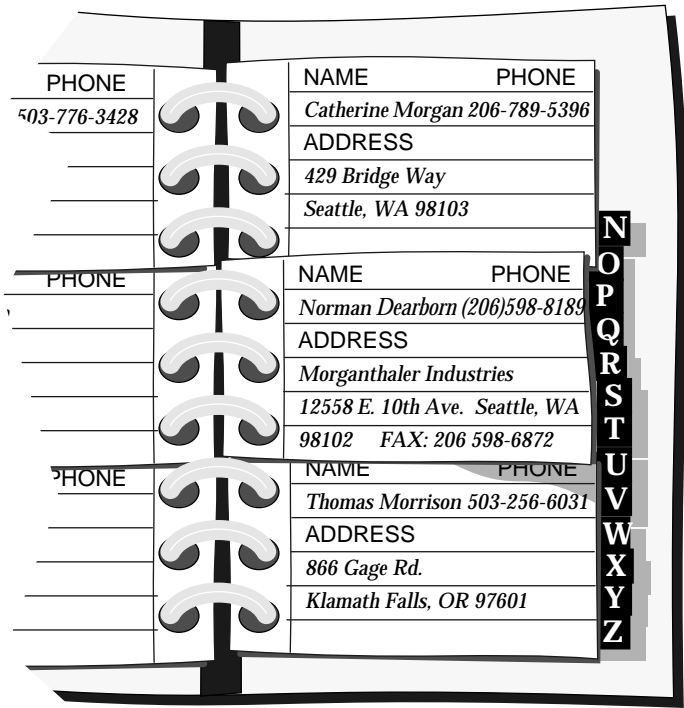


Figure 8-1 Part of a page from a telephone directory

The physical form of the existing data can be misleading. Do not let the layout of pages and entries in the telephone directory mislead you into trying to specify an entity that represents one entry in the book—some kind of alphabetized record with fields for name, number, and address. Remember it is not the medium you want to model, it is the data.

At first glance, the entities recorded in a telephone directory include the following items:

- Names (of persons and organizations)
- Addresses
- Telephone numbers

Do these entities meet the earlier criteria? They are clearly significant to the model and are generic.

Are they fundamental? A good test is to ask if an entity can vary in number independently of any other entity. After thinking about it, you realize that a telephone directory sometimes lists people who have no number or current address (people who move or change jobs). A telephone directory also can list both addresses and numbers that are used by more than one person. All three of these entities can vary in number independently; that strongly suggests they are fundamental, not dependent.

Are they unitary? Names can be split into personal names and corporate names. After thinking about it, you decide that all names should have the same features in this model; that is, you do not plan to record different information about a company than you would about a person. Likewise, you decide only one kind of address exists; no need exists to treat home addresses differently from business ones.

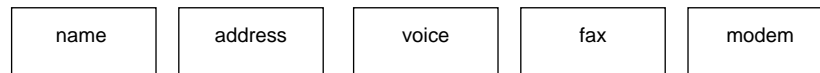
However, you also realize that more than one kind of telephone number exists, three. *Voice* numbers are answered by a person, *fax* numbers connect to a fax machine, and *modem* numbers connect to a computer. You decide that you want to record different information about each kind of number, so these three are different entities.

For the personal telephone-directory example, you decide that you want to keep track of the following entities:

- Name
- Address (mailing)
- Telephone number (voice)
- Telephone number (fax)
- Telephone number (modem)

Diagramming Your Entities

A section in this chapter will teach you how to use the entity-relationship diagrams. For now, create a separate, rectangular box for each entity in the telephone-directory example. You will learn how to put the entities together with relationships in “Diagramming Your Data Objects” on page 8-16.



Defining the Relationships

After you choose your entities, you need to consider the relationships between them. Relationships are not always obvious, but all of the ones worth recording must be found. The only way to ensure that all the relationships are found is to exhaustively list all possible relationships. You must consider every pair of entities *A* and *B* and ask, “What is the relationship between an *A* and a *B*?”

A relationship is an association between two entities. Usually, a verb or preposition connecting two entities will imply a relationship. A relationship between entities is described in terms of *connectivity*, *existence dependency*, and *cardinality*.

Connectivity

Connectivity refers to the number of entity instances. An entity instance is a particular occurrence of an entity. The three types of connectivity are one-to-one (written 1:1), one-to-many (written 1:n), and many-to-many (written m:n) shown graphically in Figure 8-2.

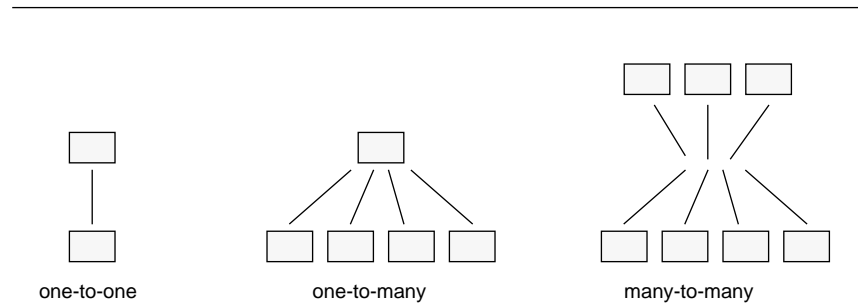


Figure 8-2 *Connectivity in relationships*

For example, in the telephone-directory example, an address can be associated with more than one name. The connectivity for the relationship between the name and address entities is one-to-many (1:n).

Existence Dependency

Existence dependency describes whether an entity in a relationship is optional or mandatory. Analyze your business rules to identify whether an entity must exist in a relationship. For example, your business rules may dictate that an address must be associated with a name. This would make the existence dependency for the relationship between the name and address entities mandatory. An optional existence dependency could be indicated by a business rule that said a person may or may not have children.

Cardinality

Cardinality places a constraint on the number of times an entity can appear in a relationship. The cardinality of a 1:1 relationship is always one. But the cardinality of a 1:n relationship is open; n could be any number. If you need to place an upper limit on n , you do it by specifying a cardinality for the relationship. For example, in a store sale example, you could limit the number of sale items that a customer can purchase at one time. You usually place cardinality constraints through your application program or through stored procedures.

For additional information about cardinality, see any entity-relationship data modeling text. References to two data modeling books are included in the Summary of this chapter.

Discover the Relationships

A compact way to discover the relationships is to prepare a matrix that names all the entities on the rows and again on the columns. Figure 8-3 is a matrix that reflects the entities for the personal telephone directory.

	name	address	number (voice)	number (fax)	number (modem)
name					
address					
number (voice)					
number (fax)					
number (modem)					

Figure 8-3 *A matrix that reflects the entities for a personal telephone directory*

You can ignore the lower triangle of the matrix, as indicated by the shaded area. You must consider the diagonal cells; that is, you must ask the question “What is the relationship between an A and another A?” In this model, the answer is always none. No relationship exists between a name and a name, or an address and another address, at least none worth recording in this model. When a relationship exists between an A and another A, you have found a recursive relationship. (See “Resolving Other Special Relationships” on page 8-28.)

For all cells for which the answer is clearly none, write `none` in the matrix. Now the matrix looks like Figure 8-4.

	name	address	number (voice)	number (fax)	number (modem)
name	none				
address		none			
number (voice)			none		
number (fax)				none	
number (modem)					none

Figure 8-4 *A matrix with initial relationships included: No entities relate to themselves*

Although in this model no entities relate to themselves, in other models this is not always true. A typical example is an employee who is the manager of another employee. Another example occurs in manufacturing, when a part entity is a component of another part.

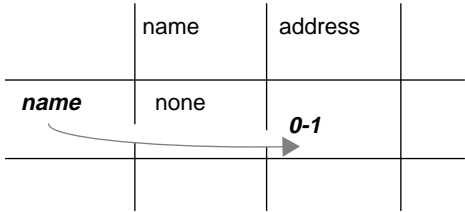
In the remaining cells, you write the connectivity relationship that exists between the entity on the row and the entity on the column. To repeat, the following kinds of relationships are possible:

- *One-to-one* (written 1:1), in which never more than one entity *A* exists for one entity *B* and never more than one *B* for one *A*.
- *One-to-many* (written 1:n), in which never more than one entity *A* exists, but there can be several entities *B* related to it (or vice versa).
- *Many-to-many* (written m:n), in which several entities *A* can be related to one *B* and several entities *B* can be related to one *A*.

One-to-many relationships are the most common. Examples of one-to-many and many-to-many relationships are shown in the telephone-directory model.

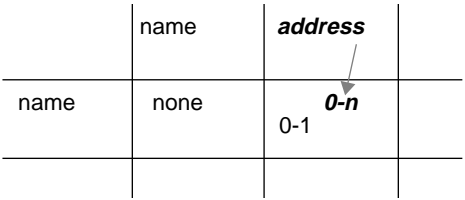
As you can see in Figure 8-4, the first unfilled cell represents the relationship between names and addresses. What connectivity lies between these entities? You can ask yourself, “how many names can be associated with an address?”

A name (you decide) can have *zero* or *one* address but no more than one. You write 0-1 opposite **name** and below **address**, as shown in the following figure:



Ask yourself how many addresses can be associated with a name. You decide that an address can be associated with more than one name. For example, you can know several people at one company or more than two people who live at the same address.

Can an address be associated with *zero* names? That is, should it be possible for an address to exist when no names use it? You decide that yes, it can. Below **address** and opposite **name**, you write 0-n, as shown in the following figure:



If you decide that an address could not exist without being associated with at least one name, you write 1-n instead of 0-n.

When the cardinality of a relationship is limited on either side to 1, it is a 1:n relationship. In this case, the relationship between names and addresses is a 1:n relationship.

Now consider the next cell, the relationship between a name and a voice number. How many voice numbers can a name be associated with, one or more than one? Glancing at your telephone directory, you see that you may often have noted more than one telephone number for a person—for some busy salesman you have a home number, an office number, a paging number,

and a car phone number. But there also can be names without associated numbers. You write 0–n opposite **name** and below **number (voice)**, as shown in the following figure:

	name	address	number (voice)
name	none	0-n 0-1	0-n

What is the other side of this relationship? How many names can be associated with a voice number? You decide that only one name can be associated with a voice number. Can a number be associated with zero names? No, you decide; no point exists for recording a number unless it is used by someone. You write 1 under **number (voice)** and opposite **name**.

	name	address	number (voice)
name	none	0-n 0-1	1 0-n

Fill out the rest of the matrix in the same fashion, using the following decisions:

- A name can be associated with more than one fax number; for example, a company can have several fax machines. Going the other way, a fax number can be associated with more than one name; for example, several people can use the same fax number.
- A modem number must be associated with exactly one name. (This is an arbitrary decree to complicate the example; pretend it is a requirement of the design.) However, a name can have more than one associated modem number; for example, a company computer can have several dial-up lines.
- Although some relationship exists between a voice number and an address, a modem number and an address, and a fax number and an

address in the real world, none needs to be recorded in this model. There already is an indirect relationship through *name*.

	name	address	number (voice)	number (fax)	number (modem)
name	none	0-1 0-n	0-n 1	0-n 1-n	0-n 1
address		none	none	none	none
number (voice)			none	none	none
number (fax)				none	none
number (modem)					none

Figure 8-5 *A completed matrix for an telephone directory*

Other decisions reflected in the matrix is that no relationship exists between a fax number and a modem number, between a voice number and a fax, or between a voice number and a modem number.

You might disagree with some of these decisions (for example, why a relationship between voice numbers and modem numbers is not supported). For the sake of our example, these are our business rules.

Diagramming Your Relationships

For now, save the matrix that you created in this section. You will learn how to create an entity-relationship diagram in “Diagramming Your Data Objects” on page 8-16.

Identifying Attributes

Entities contain *attributes*, which are characteristics or modifiers, qualities, amounts, or features. An attribute is a fact or nondecomposable piece of information about an entity. Later, when you represent an entity as a table, its attributes are added to the model as new columns.

How to Identify an Attribute

Before you can identify your attributes, you must identify your entities. After you determine your entities, ask yourself, “what characteristics do you need to know about each entity?” For example, in an *address* entity, you probably need information about *street*, *city*, and *zipcode*. Each of these characteristics of the *address* entity become attributes.

Selecting Attributes for Your Entities

In selecting attributes, choose ones that have the following qualities:

- They are significant.

Include only attributes that are useful to the users of the database.

- They are direct, not derived.

An attribute that can be derived from existing attributes—for instance, through an expression in a SELECT statement—should not be made part of the model. The presence of derived data greatly complicates the maintenance of a database.

At a later stage of the design (discussed in Chapter 10, “Tuning Your Data Model”), you can consider adding derived attributes to improve performance, but at this stage you should exclude them.

- They are nondecomposable.

An attribute can contain only single values, never lists or repeating groups. Composite values must be broken into separate attributes.

- They contain data of the same type.

For example, you would want to enter only date values in a birthday attribute, not names or telephone numbers.

The rules for defining attributes are the same as those for defining columns. For more information about defining columns, see “Placing Constraints on Columns” on page 8-21.

The following attributes are added to the telephone-directory example to produce the diagram shown in Figure 8-10 on page 8-19:

- Street, city, state, and zip code are added to the *address* entity.
- Birth date is added to the *name* entity. Also added to the *name* entity are e-mail address, anniversary date, and children’s first names.
- Type is added to the *voice* entity to distinguish car phones, home phones, and office phones. A voice number can be associated with only one voice type.

- The hours a fax machine is attended is added to the *fax* entity.
- Whether a modem supports 300-, 1,200-, or 2,400-baud rates is added to the *modem* entity.

Listing Your Attributes

For now, simply list the attributes for the telephone-directory example with the entities with which you think they belong. Your list should look something like Figure 8-6:

name	address	voice	fax	modem
fname lname bdate anniv email child1 child2 child3	street city state zipcode	vce_num vce_type	fax_num operfrom opertill	mdm_num b300 b1200 b2400

Figure 8-6 Attributes for the telephone-directory example

About Entity Occurrences

An additional data object that you need to know about is the entity occurrence. Each row in a table represents a specific, single occurrence of the entity. For example, if *customer* is an entity, a **customer** table represents the idea of customer; in it, each row represents one specific customer, such as Sue Smith. Keep in mind that entities will become tables, attributes will become columns, and rows will become entity occurrences.

Diagramming Your Data Objects

At this point, you have already discovered and understood the entities and relationships in your database. That is the most important part of the relational database design process. Once you have determined the entities and relationships, you might find it helpful to have a method for displaying your thought process during database design.

Most data modeling methods provide some form of graphically displaying your entities and relationships. Informix uses the entity-relationship diagram approach originally developed by C. R. Bachman. Entity-relationship diagrams serve the following purposes:

- They model the information needs of an organization.
- They identify entities and their relationships.
- They provide a starting point for data definition (data flow diagrams).
- They provide an excellent source of documentation for application developers as well as database and system administrators.
- They create a logical design of the database which can be translated into a physical schema.

Several different styles of documenting entity-relationship diagrams exist. If you already have a style that you prefer, use it. A sample entity-relationship diagram is represented by the following symbols:

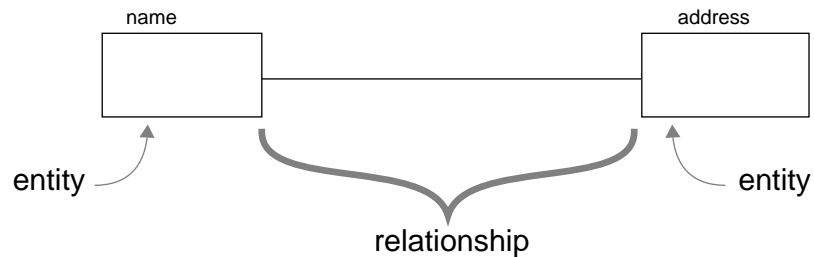


Figure 8-7 *Symbols of an entity-relationship diagram*

Entities are represented by a box. Relationships are represented by a line connecting the entities. In addition, you use several graphical items to display the following features of relationships as shown in Figure 8-8:

- A circle across a relationship link indicates *optionality* in the relationship (zero instances).
- A small bar across a relationship link indicates that *exactly one* instance of the entity is associated with another entity. (Consider the bar as a “1”.)
- The “crow’s feet” represents *many* in your relationship.

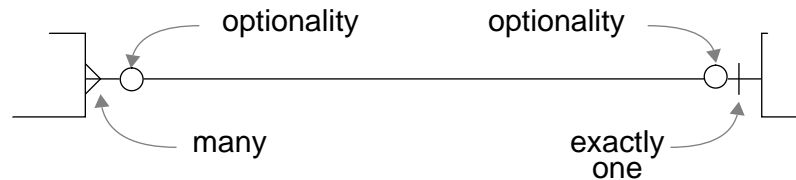


Figure 8-8 *The parts of a relationship in an entity-relationship diagram*

How You Read Entity-Relationship Diagrams

You read the diagrams first from left to right and then from right to left. In the case of the *name-address* relationship diagrammed in Figure 8-9, you read the relationships as follows: names can be associated with zero or exactly one address; addresses can be associated with zero, one, or many names.

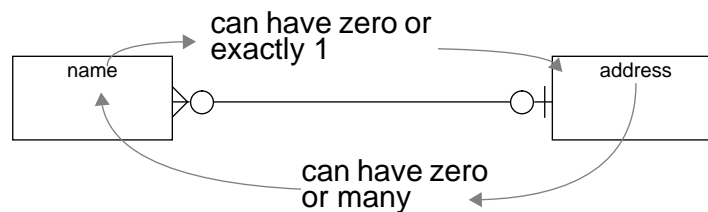


Figure 8-9 *Reading an entity-relationship diagram*

The Telephone-Directory Example

Figure 8-10 shows the telephone-directory example and includes the entities, relationships, and attributes. This diagram includes the relationships that were established with the matrix. After studying the diagram symbols, compare the entity-relationship diagram in Figure 8-10 with the matrix in Figure 8-5 on page 8-14. Verify for yourself that the relationships are the same in both figures.

A matrix such as Figure 8-5 is a useful tool when you are first designing your model because, in filling it out, you are forced to think of every possible relationship. However, the same relationships appear in a diagram such as Figure 8-10, and this type of diagram may be easier to read when you are reviewing an existing model.

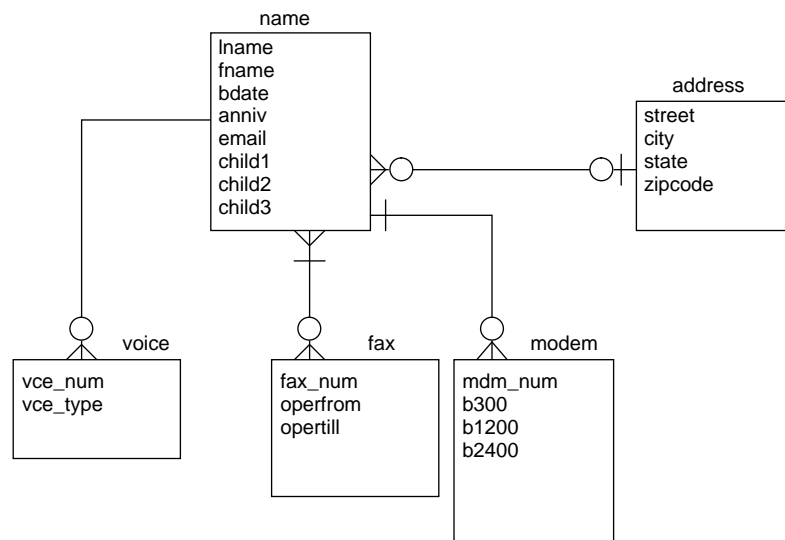


Figure 8-10 Preliminary entity-relationship diagram of the Telephone-Directory Example

What's Next

Subsequent sections describe the following tasks:

- How to translate the entities, relationships, and attributes into relational constructs
- How to resolve the entity-relationship data model
- How to normalize the entity-relationship data model

Chapter 9, "Implementing Your Data Model," shows you how to create a database from the entity-relationship data model.

Translating E-R Data Objects Into Relational Constructs

All the data objects you have learned about so far—entities, relationships, attributes, and entity occurrences—will be translated into SQL tables, joins between tables, columns, and rows. The tables, columns, and rows of your database must fit the rules found in “Rules for Defining Tables, Rows, and Columns.”

Your data objects should fit these rules before you normalize your data objects. Normalizing your data objects is accomplished by analyzing the dependencies between your entities, relationships, and attributes. Normalizing is discussed in “Normalizing Your Data Model” on page 8-29.

After you normalize the data model, you can use SQL statements to create a database based on your data model. Chapter 9, “Implementing Your Data Model,” describes how to create your database and provides the database schema for the telephone-directory example.

Each entity you choose is represented as a table in the model. The table stands for the entity as an abstract concept, and each row represents a specific, individual *occurrence* of the entity. In addition, each attribute of an entity is represented by a column in the table.

The following ideas are fundamental to most relational data model methods, including the entity-relationship data model. By following these rules while designing your data model, you will save time and effort when you normalize your model.

Rules for Defining Tables, Rows, and Columns

You are already familiar with the idea of a *table* composed of *rows* and *columns*. But you must respect the following rules while you define the tables of a formal data model:

- Rows must stand alone.

Each row of a table is independent and does not depend on any other row of the *same* table. As a consequence, the order of the rows in a table is not significant in the model. The model should still be correct even if all the rows of a table are shuffled into random order.

After the database is implemented, you can tell the database server to store rows in a certain order for the sake of efficiency, but that does not affect the model.

- Rows must be unique.
In every row, some column must contain a unique value. If no single column has this property, the values of some group of columns taken as a whole must be different in every row.
- Columns must stand alone.
The order of columns within a table has no meaning in the model. The model should still be correct even if the columns are rearranged.
After the database is implemented, programs and stored queries that use an asterisk to mean *all columns* are dependent on the final order of columns, but that does not affect the model.
- Column values must be unitary.
A column can contain only single values, never lists or repeating groups. Composite values must be broken into separate columns. For example, if you decide to treat a person's first and last names as separate values, as shown in the examples in this chapter, they must be in separate columns, not in a single *name* column.
- Each column must have a unique name.
Two columns within the same table cannot share the same name. However, you may have columns that contain similar information. For example, the name table in the telephone-directory example contains columns for children's names. You can name each column, **child1**, **child2**, and so on.
- Each column must contain data of a single type.
A column must contain information of the same data type. For example, a column identified as an integer must contain only numeric information, not characters from a name.

If your previous experience is only with data organized as arrays or sequential files, these rules might seem unnatural. However, relational database theory shows that you can represent all types of data using only tables, rows, and columns that follow these rules. With a little practice, these rules become automatic.

Placing Constraints on Columns

When you define your table and columns with the `CREATE TABLE` statement, you constrain each column. These constraints specify whether you want the column to contain characters or numbers, the form that you want dates to

use, and other constraints. A *domain* describes the constraints on and identifies the set of valid values attributes may assume. The domain characteristics of a column can consist of the following items:

- Data type (INTEGER, CHAR, DATE, and so on)
- Format (for example, yy/mm/dd)
- Range (for example, 1,000-5,400)
- Meaning (for example, personnel number)
- Allowable values (for example, only grades S or U)
- Uniqueness
- Null support
- Default value
- Referential constraints

You define the domain characteristics when you create your tables. Defining domains and creating your tables and database are discussed in Chapter 9, “Implementing Your Data Model.”

Determining Keys for Tables

The columns of a table are either *key* columns or *descriptor* columns. A key column is one that uniquely identifies a particular row in the table. For example, a social-security number is unique for each employee. A descriptor column specifies the nonunique characteristics of a particular row in the table. For example, two employees may have the same first name, Sue. The first name Sue is a nonunique characteristic of an employee. The main types of keys in a table are: primary keys and foreign keys.

You designate primary and foreign keys when you create your tables. Primary and foreign keys are used to relate tables together physically. Your next task is to specify a primary key for each table. That is, you must identify some quantifiable characteristic of the table that distinguishes each row from every other.

Primary Keys

The *primary key* of a table is the column whose values are different in every row. Because they are different, they make each row unique. If no one such column exists, the primary key is a *composite* of two or more columns whose values, taken together, are different in every row.

Every table in the model must have a primary key. This follows automatically from the rule that all rows must be unique. If necessary, the primary key is composed of all the columns taken together.

The primary key should be a numeric data type (INT or SMALLINT), SERIAL data type, or a short character string (as used for codes). It is recommended that you avoid using long character strings as primary keys.

Null values are never allowed in a primary-key column. Null values are not comparable; that is, they cannot be said to be alike or different. Hence, they cannot make a row unique from other rows. If a column permits null values, it cannot be part of a primary key.

What is a Null Value?

Columns in a table may be designated as containing null values. This means that the value for the column may be unknown or not applicable. For example, in the telephone-directory example, the **anniv** column of the *name* table may contain null values; if you do not know the person's anniversary, you do not specify it. Be careful not to confuse null value with zero or blank value.

Some entities have ready-made primary keys such as catalog codes or identity numbers, which are defined outside the model. These are user-assigned keys.

Sometimes more than one column or group of columns can be used as the primary key. All columns or groups that qualify to be primary keys are called *candidate keys*. All candidate keys are worth noting because their property of uniqueness makes them predictable in a SELECT operation. When you select the columns of a candidate key, you know the result can contain no duplicate rows. Sometimes that allows you to predict the uniqueness of the rows returned. It always means that the result of a SELECT operation can be a table in its own right, with the selected candidate key as its primary key.

Composite Keys

Some entities lack features that are reliably unique. Different people can have identical names; different books can have identical titles. You can usually find a composite of attributes that work as a primary key. For example, it is rare for people to have identical names and identical addresses or for different books to have identical titles, authors, and publication dates.

System-Assigned Keys

A system-assigned primary key is usually preferable to a composite key. A system-assigned key is a number or code that is attached to each instance of an entity when it is first entered into the database. The easiest system-assigned keys to implement are serial numbers because the database server can generate them automatically. Informix offers the `SERIAL` data type for serial numbers. However, the people who use the database might not like a plain numeric code. Other codes can be based on actual data; for example, an employee identification code could be based on the person's initials combined with the digits of the date they were hired. In the telephone-directory example, a system-assigned primary key is used for the *name* table.

Foreign Keys (Join Columns)

A *foreign key* is simply a column or group of columns in one table that contains values that match the *primary key* in another table. Foreign keys are used to join tables; in fact, most of the *join columns* referred to earlier in this book are foreign-key columns. Figure 8-11 shows the primary and foreign keys of the *customer* and *order* tables from the **stores6** database.

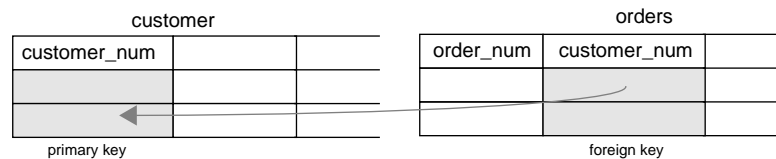


Figure 8-11 Primary-foreign keys in the customer-order relationships

Foreign keys are noted wherever they appear in the model because their presence can restrict your ability to delete rows from tables. Before you can delete a row safely, either you must delete all rows that refer to it through foreign keys or you must define the relationship using special syntax that allows you to delete rows from primary-key and foreign-key columns with a single delete command. The database server disallows deletes that violate referential integrity.

You can always preserve referential integrity by deleting all foreign-key rows before you delete the primary key to which they refer. If you are imposing referential constraints on your database, the database server does not permit you to delete primary keys with matching foreign keys. It also does not

permit you to add a foreign-key value that does not reference an existing primary-key value. Referential integrity is discussed in Chapter 4, “Modifying Data.”

Adding Keys to the Telephone-Directory Diagram

The initial choices of primary and foreign keys is shown in Figure 8-12. This diagram reflects some important decisions.

For the *name* table, the primary key **rec_num** is chosen. Note that the data type for **rec_num** is SERIAL. The values for **rec_num** are system-generated. If you look at the other columns (or attributes) in the *name* table, you see that the data types associated with the columns are mostly character-based. None of these columns alone are good candidates for a primary key. If you combine elements of the table into a composite key, you create an exceedingly cumbersome key. By using a SERIAL data type, you have a key that you can also use to join the other tables to the *name* table.

For the *voice*, *fax*, and *modem* tables, the telephone numbers are shown as primary keys. Note that these tables are joined to the *name* table through the **rec_num** key.

The *address* table also uses a system-generated primary key, **id_num**. The *address* table must have a primary key because the business rules state that an address can exist when no names use it. If the business rules prevent an address from existing unless a name is associated with it, then the *address* table could be joined to the *name* table with the foreign key **rec_num** only.

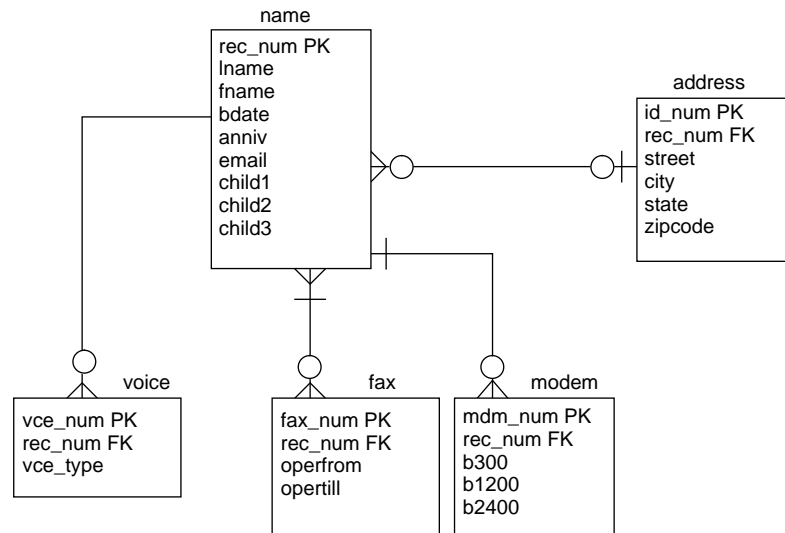


Figure 8-12 Telephone-directory diagram with primary and foreign keys added

Resolving Your Relationships

The aim of a good data model is to create a structure that provides the database server with quick access. The telephone-directory data model can be further refined by resolving the relationships and normalizing the data model. This section addresses the hows and whys of resolving your relationships. Normalizing your data model is discussed in “Normalizing Your Data Model” on page 8-29.

Resolving m:n Relationships

Many-to-many relationships add complexity and confusion to your model and to the application development process. The key to resolving many-to-many relationships is to separate the two entities and create two one-to-many relationships between them with a third *intersect* entity. The intersect entity usually contains attributes from both connecting entities.

To resolve a many-to-many relationship, analyze your business rules again. Have you accurately diagrammed the relationship? In the telephone-directory example, we have a many-to-many relationship between the *name* and *fax* entities as shown in Figure 8-12 on page 8-26. To resolve the relationship between *name* and *fax*, we carefully reviewed the business rules. The business rules say: “One person can have zero, one, or many fax numbers; a *fax number can be for several people*.” Based on what we selected earlier as our primary key for the voice entity, a many-to-many relationship exists.

A problem exists in the *fax* entity because the telephone number, which is designated as the primary key, can appear more than one time in the *fax* entity; this violates the qualification of a primary key. Remember, the primary key must be unique.

This many-to-many relationship is resolved by adding an intersect entity between *name* and *fax* entities. The new intersect entity, *faxname*, contains two attributes, **fax_num** and **rec_num**. The primary key for the entity is a composite of both attributes. Individually, each attribute is a foreign key that references the table from which it came. The relationship between the *name* and *faxname* tables is one-to-many because one name can be associated with many fax numbers; in the other direction, each *faxname* combination can be associated with one **rec_num**. The relationship between the *fax* and *faxname* tables is one-to-many because each number can be associated with many *faxname* combinations.

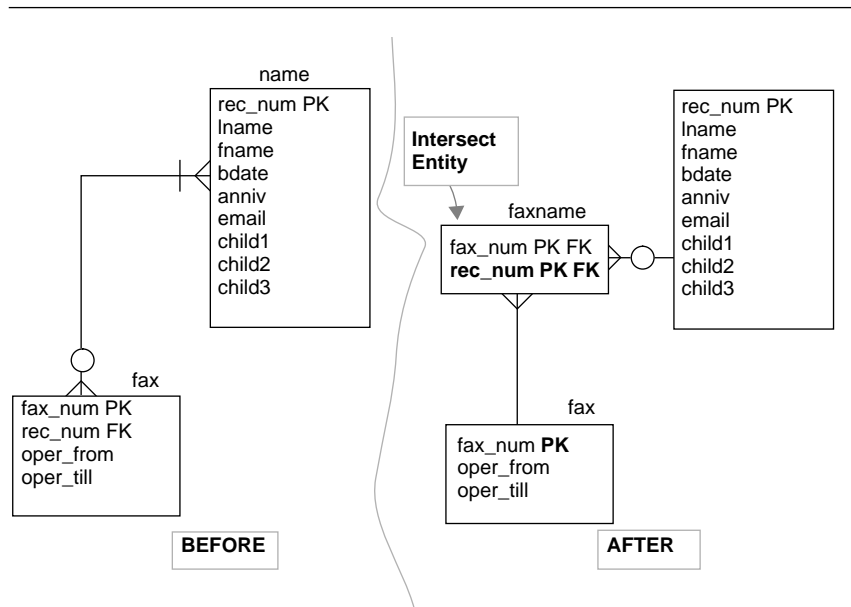


Figure 8-13 Resolving a many-to-many relationship

Resolving Other Special Relationships

You may encounter other special relationships that can hamper a smooth running database. These relationships are shown in the following list:

- Complex relationships
- Recursive relationships
- Redundant relationships

A *complex* relationship is an association among three or more entities. All the entities must be present for the relationship to exist. To reduce this complexity, reclassify all complex relationships as an entity, related through binary relationships to each of the original entities.

A *recursive* relationship is an association between occurrences of the same entity type. These types of relationships do not occur often. Examples of recursive relationships are bill-of-materials (parts are composed of subparts) and organizational structures (employee manages employees). See Chapter 5, “Programming with SQL,” for an extended example of a recursive relationship. You may choose not to resolve recursive relationships.

A *redundant* relationship is when two or more relationships are used to represent the same concept. Redundant relationships add complexity to the data model and may lead a developer to incorrectly place attributes in the model. Redundant relationships may appear as duplicated entries in your entity-relationship diagram. For example, you may have two entities that contain the same attributes. To resolve a redundant relationship, review your data model. Do you have more than one entity that contains the same attributes? You may need to add an entity to the model to resolve the redundancy. Chapter 10, “Tuning Your Data Model,” discusses additional topics related to redundancy in a data model

Normalizing Your Data Model

The telephone-directory example shown in this chapter appears to be a good model. It could be implemented at this point into a database, but it might present problems later on with application development and data-manipulation operations. *Normalization* is a formal approach to applying a set of rules used in associating attributes with entities.

Normalizing your data model can do the following things:

- Produce greater flexibility in your design
- Ensure that attributes are placed in the proper tables
- Reduce data redundancy
- Increase programmer effectiveness
- Lower application maintenance costs
- Maximize stability of the data structure

Normalization consists of several steps to reduce the entities to more desirable physical properties. These steps are called normalization rules, also referred to as *normal forms*. Several normal forms exist; this chapter discusses the first three normal forms. Each normal form constrains the data to be more organized than the last form. Because of this, you must achieve first normal form before you can achieve second normal form, and you must achieve second normal form before you can achieve third normal form.

First Normal Form

An entity is in first normal form if it contains no repeating groups. In relational terms, this means that a table is in first normal form if it contains no repeating columns. Repeating columns make your data less flexible, waste

disk space, and make it more difficult to search for data. In the telephone-directory example, it appears that the *name* table contains repeating columns, *child1*, *child2*, and *child3*, as shown in Figure 8-14:

name								
rec_num	lname	fname	bdate	anniv	email	child1	child2	child3

repeating columns

Figure 8-14 *Unnormalized name entity*

Some problems can be noted in the current table. Space is always reserved on the disk for three child records, whether the person has children or not. The maximum number of children that you could record is three; but some of your acquaintances might have four or more children. If you look for a particular child, you would have to search all three columns in every row.

You can eliminate the repeating columns and bring the table to first normal form by separating the table into two tables as shown in Figure 8-15. Put the repeating columns into one of the tables. The association between the two tables is established with a primary and foreign key combination. Because a child cannot exist without an association in the *name* table, we can reference the name table with a foreign key, **rec_num**.

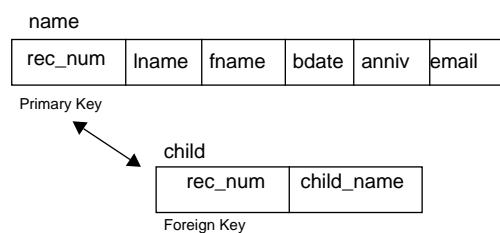


Figure 8-15 *First normal form reached for name entity*

In addition, the *modem-name* relationship is not at first normal form because the columns b300, b1200, and b2400 are considered repeating columns. This relationship is normalized in a similar manner as the *name-child* relationship,

as shown in Figure 8-16. A **b_type** attribute is added to the entity which can contain occurrences of b300, b1200, and b2400. Figure 8-16 shows the data model normalized through first normal form.

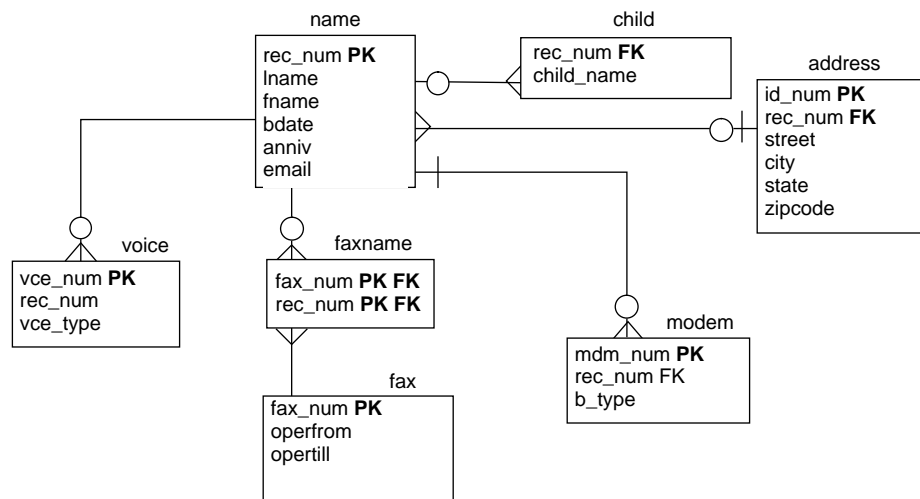


Figure 8-16 The data model of a personal telephone directory

Second Normal Form

An entity is in second normal form if it is in first normal form and all its attributes depend on the whole (primary) key. In relational terms, this means that every column in a table must be *functionally dependent* on the whole primary key of that table. Functional dependency indicates that a link exists between the values in two different columns.

To say that the value of an attribute *depends on* a column means that, if the value in the column changes, the value of the attribute must also change. The attribute is a function of the column. The following explanations make this more specific:

- If the table has a one-column primary key, the attribute must depend on that key.
- If the table has a composite primary key, the attribute must depend on the values in all of its columns taken as a whole, not on just one or some of them.

- If the attribute also depends on other columns, these must be columns of a candidate key, that is, columns that are unique in every row.

If you do not convert your model to second normal form, you risk data redundancy and difficulty in changing data. Convert first normal form tables to second normal form tables by removing columns that are not dependent on the whole primary key.

Third Normal Form

An entity is in third normal form if it is in second normal form and all its attributes are nontransitively dependent on the primary key. *Transitive dependence* means that descriptor key attributes depend not only on the whole primary key but also on other descriptor key attributes that, in turn, depend on the whole primary key. In SQL terms, third normal form means that no column within a table is dependent on a descriptor column that, in turn, depends on the primary key.

To convert to third normal form, remove attributes that depend on other descriptor key attributes.

Summary of Normalization Rules

The following normal forms are discussed in this section:

- First normal form: A table is in first normal form if it contains no repeating columns.
- Second normal form: A table is in second normal form if it is in first normal form and contains only columns that are dependent on the whole primary key.
- Third normal form: A table is in third normal form if it is in second normal form and contains only columns that are nontransitively dependent on the primary key.

When you follow these rules, the tables of the model are in what E.F. Codd, the inventor of relational databases, calls *third-normal form*. When tables are not in third-normal form, either redundant data exists in the model or problems exist when you attempt to update the tables.

If you cannot find a place for an attribute that observes these rules, then you have probably made one of the following errors:

- The attribute is not well-defined.
- The attribute is derived, not direct.
- The attribute is really an entity or a relationship.

- Some entity or relationship is missing from the model.

Summary

This chapter summarized and illustrated the following steps of entity-relationship data modeling:

1. *Identify and define* your principal data objects, including the following options:
 - Entities
 - Relationships
 - Attributes
2. *Diagram* your data objects using the entity-relationship diagram approach.
3. *Translate* your entity-relationship data objects into relational constructs.
 - Determine the primary and foreign keys for each entity.
4. *Resolve* your relationships, particularly the following relationships:
 - 1:1 relationships
 - m:n relationships
 - Other special relationships
5. *Normalize* your data model in one of the following forms:
 - First normal form
 - Second normal form
 - Third normal form

When the process is done right, you must examine every aspect of the data not once, but several times.

If you are interested in learning more about database design, you can attend the Informix course, *Relational Database Design*. This thorough course presents how to create an entity-relationship data model.

If you are interested in pursuing more about database design on your own, the following excellent books are recommended:

- *Database Modeling and Design, The Entity-Relationship Approach*, by Toby J. Teorey, Morgan Kaufman Publishers, Inc., 1990
- *Handbook of Relational Database Design*, by Candace C. Fleming and Barbara von Halle, Addison-Wesley Publishing Company, 1989

Implementing Your Data Model

Chapter Overview	3
Defining the Domains	3
Data Types	4
Choosing a Data Type	4
Numeric Types	7
Chronological Types	12
Character Types	15
Changing the Data Type	19
Default Values	19
Check Constraints	20
Creating the Database	20
Using CREATE DATABASE	21
Using CREATE DATABASE with INFORMIX-OnLine Dynamic Server	21
Using CREATE DATABASE with Other Informix Database Servers	23
Using CREATE TABLE	24
Using Command Scripts	26
Capturing the Schema	26
Executing the File	26
An Example	26
Populating the Tables	27
Summary	29

Chapter Overview

Once a data model is prepared, it must be implemented as a database and tables. This chapter covers the decisions that you must make to implement the model.

The first step in implementation is to complete the data model by defining a domain, or set of data values, for every column. The second step is to implement the model using SQL statements.

The first section of this chapter covers defining domains in detail. The second section shows how you create the database (using the CREATE DATABASE and CREATE TABLE statements) and populate it with data.

Defining the Domains

To complete the data model described in Chapter 8, “Building Your Data Model,” you must define a domain for each column. The domain of a column describes the constraints on and identifies the set of valid values attributes (or columns) may assume.

The purpose of a domain is to guard the *semantic integrity* of the data in the model; that is, to ensure that it reflects reality in a sensible way. If a name can be entered where a telephone number was planned or a fractional number where an integer should be, the integrity of the data model is at risk.

You define a domain by first defining the *constraints* that a data value must satisfy before it can be part of the domain. Column domains are specified using the following constraints:

- Data types
- Default values
- Check constraints

In addition, referential constraints can be placed on columns by identifying the primary and foreign keys in each table. How to identify these keys was discussed in Chapter 8, “Building Your Data Model.”

Data Types

The first constraint on any column is the one that is implicit in the data type for the column. When you choose a data type, you constrain the column so that it contains only values that can be represented by that type.

Each data type represents certain kinds of information and not others. The correct data type for a column is the one that represents all the data values that are proper for that column but as few as possible of the values that are not proper for it.

Choosing a Data Type

Every column in a table must have a data type chosen from the types that the database server supports. The choice of data type is important for the following reasons:

- It establishes the basic domain of the column; that is, the set of valid data items that the column can store.
- It determines the kinds of operations you can perform on the data. For example, you cannot apply aggregate functions, such as SUM, to columns with a character data type.
- It determines how much space each data item occupies on disk. This is not important for small tables, but if a table has tens or hundreds of thousands of rows, the difference between a 4-byte and an 8-byte type can be crucial.

Using Data Types in Referential Constraints

Almost all data type combinations must match. For example, if a primary key is defined as CHAR, the foreign key must also be defined as CHAR. However, when you specify a SERIAL data type on a primary key in one table, you specify an INTEGER on the foreign key of the relationship. The SERIAL and INTEGER construction is the only data type combination that you can mix in a relationship.

The decision tree shown in Figure 9-1 summarizes the choices among data types. They are explained in the following sections.

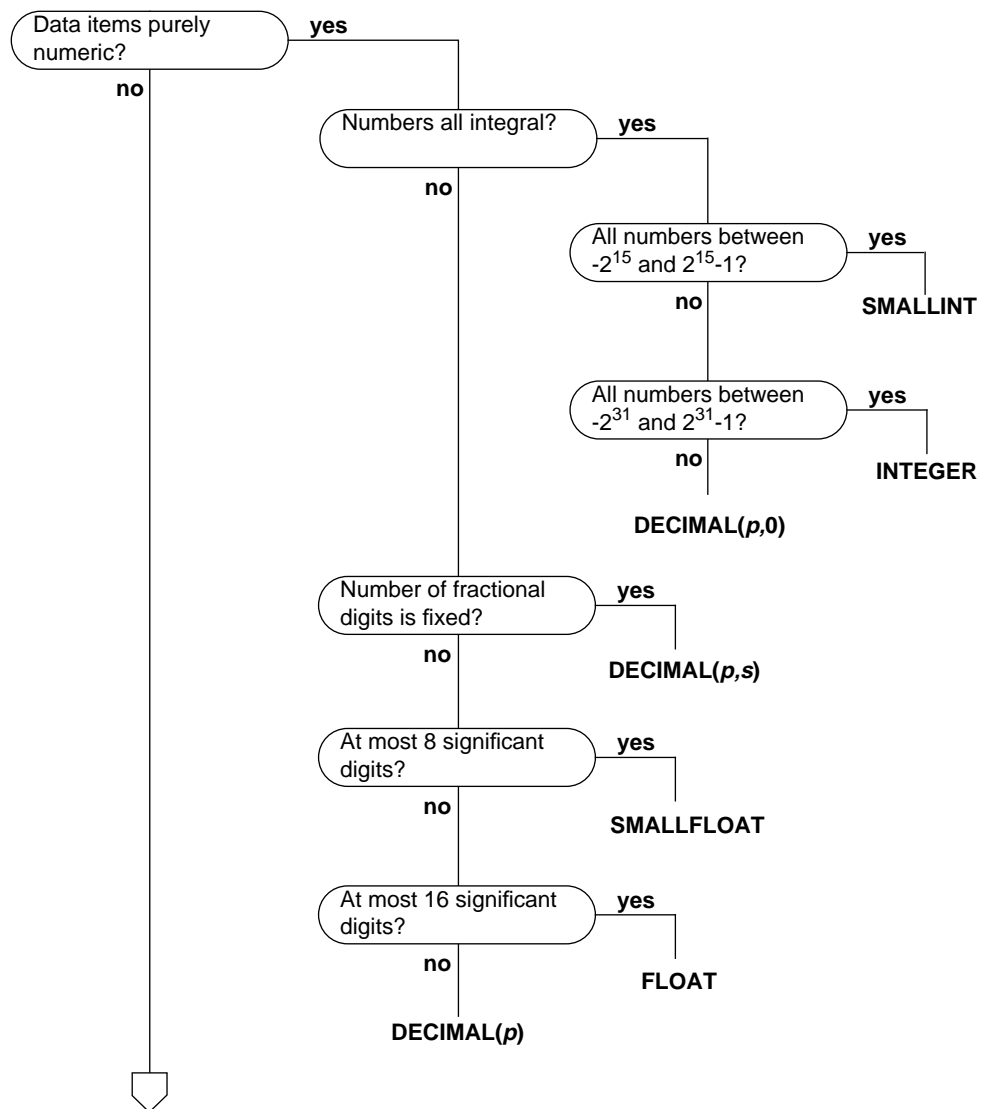


Figure 9-1 A diagram of the decisions to be made in choosing a data type (1 of 2)

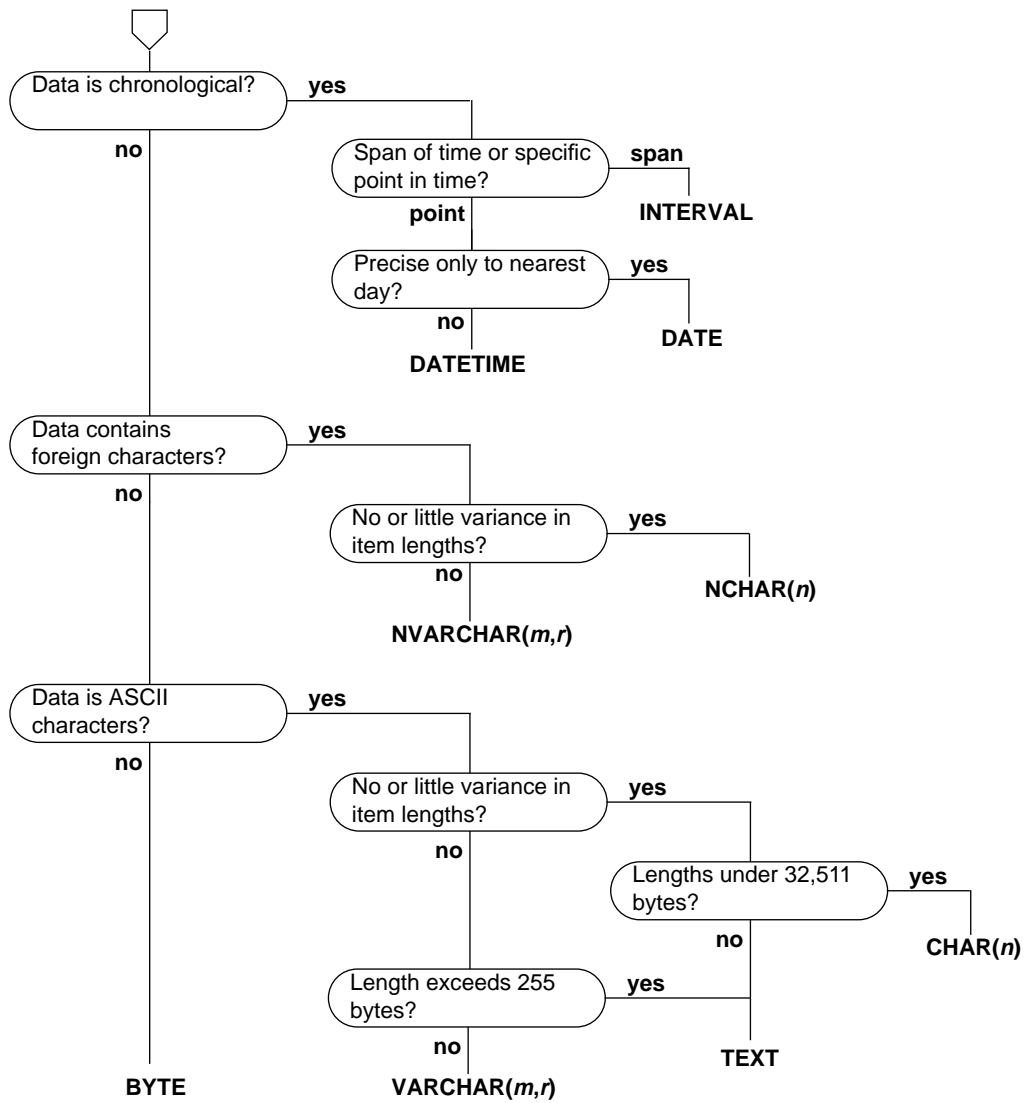


Figure 9-1 A diagram of the decisions to be made in choosing a data type (2 of 2)

Numeric Types

Informix database servers support eight numeric data types. Some are best suited for counters and codes, some for engineering quantities, and some for money.

Counters and Codes: INTEGER and SMALLINT

The INTEGER and SMALLINT data types hold small whole numbers. They are suited for columns that contain counts, sequence numbers, numeric identity codes, or any range of whole numbers when you know in advance the maximum and minimum values to be stored.

Both types are stored as signed binary integers. INTEGER values have 32 bits and can represent whole numbers from -2^{31} through $2^{31}-1$; that is, from -2,147,483,647 through 2,147,483,647. (The maximum negative number, -2,147,483,248, is reserved and cannot be used.)

SMALLINT values have only 16 bits. They can represent whole numbers from -32,767 through 32,767. (The maximum negative number, -32,768, is reserved and cannot be used.)

These data types have the following advantages:

- They take up little space (2 bytes per value for SMALLINT and 4 bytes per value for INTEGER).
- Arithmetic expressions such as SUM and MAX as well as sort comparisons can be done very efficiently on them.

The disadvantage to using INTEGER and SMALLINT is the limited range of values they can store. The database server does not store a value that exceeds the capacity of an integer. Of course, this is not a problem when you know the maximum and minimum values to be stored.

Automatic Sequences: SERIAL

The SERIAL data type is simply INTEGER with a special feature. Whenever a new row is inserted into a table, the database server automatically generates a new value for a SERIAL column. A table can have only one SERIAL column. Because the database server generates them, the serial values in new rows are always different even when multiple users are adding rows at the same time. This is a useful service, because it is quite difficult for an ordinary program to coin unique numeric codes under those conditions.

How Many Serialized Rows?

After inserting 2^{31} rows in a table, the database server uses up all the positive serial numbers. Should you be concerned about this? Probably not because to make this happen you need to insert a row every second for 68 years. However, if it did occur, the database server would continue generating new numbers. It would treat the next-serial quantity as a signed integer. Because it uses only positive values, it would simply wrap around and start generating integer values beginning with 1.

The sequence of generated numbers always increases. When rows are deleted from the table their serial numbers are not reused. This means that rows sorted on a SERIAL column are returned in the order in which they were created. That cannot be said of any other data type.

You can specify the initial value in a SERIAL column in the CREATE TABLE statement. This makes it possible to generate different subsequences of system-assigned keys in different tables. The **stores6** database uses this technique. In **stores6**, the customer numbers begin at 101, and the order numbers start at 1001. As long as this small business does not register more than 899 customers, all customer numbers have three digits and order numbers have four.

A SERIAL column is not automatically a unique column. If you want to be perfectly sure no duplicate serial numbers occur, you must apply a unique constraint (see “Using CREATE TABLE” on page 9-24). However, if you define the table using the interactive schema editor in **DB-Access** or **INFORMIX-SQL**, it automatically applies a unique constraint to any SERIAL column.

The SERIAL data type has the following advantages:

- It provides a convenient way to generate system-assigned keys.
- It produces unique numeric codes even when multiple users are updating the table.
- Different tables can use different ranges of numbers.

It has the following disadvantages:

- Only one SERIAL column is permitted in a table.
- It can produce only arbitrary numbers (then again, arbitrary numeric codes might not be acceptable to the database users).

Altering the Next SERIAL Number

The starting value for a SERIAL column is set when the column is created (see “Using CREATE TABLE” on page 9-24). You can use the ALTER TABLE statement later to reset the *next* value, the value that is used for the next-inserted row.

You cannot set the *next* value below the current maximum value in the column because that can lead the database server to generate duplicate numbers. However, you can set the *next* value to any value higher than the current maximum, thus creating gaps in the sequence.

Approximate Numbers: FLOAT and SMALLFLOAT

In scientific, engineering, and statistical applications, numbers are often known to only a few digits of accuracy, and the magnitude of a number is as important as its exact digits.

The floating-point data types are designed for these applications. They can represent any numerical quantity, fractional or whole, over a wide range of magnitudes from the cosmic to the microscopic. For example, they can easily represent both the average distance from the Earth to the Sun (1.5×10^9 meters) or Planck’s constant (6.625×10^{-27}). Their only restriction is their limited precision. Floating-point numbers retain only the most significant digits of their value. If a value has no more digits than a floating-point number can store, the value is stored exactly. If it has more digits, it is stored in approximate form, with its least-significant digits treated as zeros.

This lack of exactitude is fine for many uses, but you should never use a floating-point data type to record money or any other quantity for which it is an error to change the least significant digits to zero.

Two sizes of floating-point data types exist. The FLOAT type is a double-precision, binary floating-point number as implemented in the C language on your computer. One usually takes up 8 bytes. The SMALLFLOAT (also known as REAL) type is a single-precision, binary floating-point number that usually takes up 4 bytes. The main difference between the two data types is their precision. A FLOAT column retains about 16 digits of its values; a SMALLFLOAT column retains only about 8 digits.

Floating-point numbers have the following advantages:

- They store very large and very small numbers, including fractional ones.
- They represent numbers compactly in 4 or 8 bytes.

- Arithmetic functions such as AVG, MIN, and sort comparisons, are efficient on these data types.

The main disadvantage of floating-point numbers is that digits outside their range of precision are treated as zeros.

Adjustable-Precision Floating Point: DECIMAL(*p*)

The DECIMAL(*p*) data type is a floating-point type similar to FLOAT and SMALLFLOAT. The important difference is that you specify how many significant digits it retains. The precision you write as *p* may range from 1 to 32, from fewer than SMALLFLOAT up to twice the precision of FLOAT.

The magnitude of a DECIMAL(*p*) number ranges from 10^{-130} to 10^{124} .

It is easy to be confused about decimal data types. The one under discussion is DECIMAL(*p*); that is, DECIMAL with only a precision specified. The size of DECIMAL(*p*) numbers depends on their precision; they occupy $1+p/2$ bytes (rounded up to a whole number, if necessary).

DECIMAL(*p*) has the following advantages over FLOAT:

- Precision can be set to suit the application, from highly approximate to highly precise.
- Numbers with as many as 32 digits can be represented exactly.
- Storage is used in proportion to the precision of the number.
- Every Informix database server supports the same precision and range of magnitudes, regardless of the host operating system.

The DECIMAL(*p*) data type has the following disadvantages:

- Arithmetic and sorting are somewhat slower than on FLOAT numbers.
- Many programming languages do not support the DECIMAL(*p*) data format the way they support FLOAT and INTEGER. When a program extracts a DECIMAL(*p*) value from the database, it may have to convert the value to another format for processing. (However, **INFORMIX-4GL** programs can use DECIMAL(*p*) values directly.)

Fixed-Point Numbers: DECIMAL and MONEY

Most commercial applications need to store numbers that have fixed numbers of digits on the right and left of the decimal point. Amounts of money are the most common examples. Amounts in U.S. and other currencies are written with two digits to the right of the decimal point. Normally,

you also know the number of digits needed on the left, depending on whose transactions are recorded—perhaps 5 digits for a personal budget, 7 for a small business, and 12 or 13 for a national budget.

These numbers are *fixed-point* numbers because the decimal point is fixed at a specific place, regardless of the value of the number. The DECIMAL(*p,s*) data type is designed to hold them. When you specify a column of this type, you write its *precision* (*p*) as the total number of digits it can store, from 1 to 32. You write its *scale* (*s*) as the number of those digits that fall to the right of the decimal point. (The relation between precision and scale is diagrammed in Figure 9-2.) Scale can be zero, meaning it stores only whole numbers. When this is done, DECIMAL(*p,s*) provides a way of storing integers of up to 32 digits.

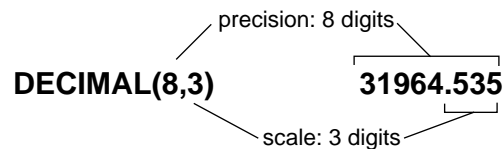


Figure 9-2 *The relation between precision and scale in a fixed-point number*

As with the DECIMAL(*p*) data type, DECIMAL(*p,s*) takes up space in proportion to its precision. One value occupies $1 + p/2$ bytes, rounded up to a whole number of bytes.

The MONEY type is identical to DECIMAL(*p,s*), but with one extra feature. Whenever the database server converts a MONEY value to characters for display, it automatically includes a currency symbol.

The advantages of DECIMAL(*p,s*) over INTEGER and FLOAT are that much greater precision is available (up to 32 digits as compared to 10 for INTEGER and 16 for FLOAT), and both the precision and the amount of storage required can be adjusted to suit the application.

The disadvantages are that arithmetic is less efficient and that many programming languages do not support numbers in this form. Therefore, when a program extracts a number, it usually must convert it to another numeric form for processing. (However, INFORMIX-4GL programs can use DECIMAL(*p,s*) and MONEY values directly.)

Whose Money? Choosing a Currency Format

Each nation has its own way of punctuating amounts of money. When an Informix database server displays a MONEY value, it refers to a currency format defined to the operating system, usually in a variable named DBMONEY. A currency symbol can precede or follow the amount, and the decimal delimiter can be set to a period, a comma, or another character. When in NLS mode, you can set the X/Open category LC_MONETARY to specify a foreign currency symbol and format. For details, see Chapter 4 of the *Informix Guide to SQL: Reference*.

Chronological Types

Informix database servers support three data types for recording time. The DATE data type stores a calendar date. DATETIME records a point in time to any degree of precision from a year to a fraction of a second. The INTERVAL data type stores a span of time; that is, a duration.

Calendar Dates: DATE

The DATE data type stores a calendar date. A DATE value is actually a signed integer whose contents are interpreted as a count of full days since midnight on December 31, 1899. Most often it holds a positive count of days into the current century.

The DATE format has ample precision to carry dates into the far future (58,000 centuries). Negative DATE values are interpreted as counts of days prior to the epoch date; that is, a DATE of -1 represents the day December 30, 1899.

Because DATE values are integers, Informix database servers permit them to be used in arithmetic expressions. For example, you can take the average of a DATE column, or you can add 7 or 365 to a DATE column. In addition, a rich set of functions exist specifically for manipulating DATE values. (See Chapter 1 of the *Informix Guide to SQL: Syntax*.)

The DATE data type is compact at 4 bytes per item. Arithmetic functions and comparisons execute quickly on a DATE column.

Exact Points in Time: DATETIME

The DATETIME data type stores any moment in time in the era beginning 1 A.D. In fact, DATETIME is really a family of 28 data types, each with a different precision. When you define a DATETIME column, you specify its precision. It may contain any sequence from the list *year*, *month*, *day*, *hour*, *minute*, *second*, and *fraction*. Thus, you can define a DATETIME column that stores only

M/D/Y? D-M-Y? Choosing a Date Format

You can punctuate and order the components of a date in many ways. When an Informix database server displays a DATE value, it refers to a format defined to the operating system, usually in a variable named DBDATE. The numbers for day, month, and year can be shown in any order, separated by a chosen delimiter. For more information, see Chapter 4 of the *Informix Guide to SQL: Reference*.

a year, only a month and day, a date and time that is exact to the hour, or even to the millisecond. The size of a DATETIME value ranges from 2 to 11 bytes depending on its precision, as shown in Figure 9-3.

The advantage of DATETIME is that it can store dates more precisely than to the nearest day, and it can store time values. Its sole disadvantage is an inflexible display format, but this can be circumvented (see “Forcing the Format of a DATETIME or INTERVAL Value” on page 9-16).

Precision	Size*	Precision	Size*
year to year	3	day to hour	3
year to month	4	day to minute	4
year to day	5	day to second	5
year to hour	6	day to fraction(<i>f</i>)	$5+f/2$
year to minute	7	hour to hour	2
year to second	8	hour to minute	3
year to fraction (<i>f</i>)	$8+f/2$	hour to second	4
month to month	2	hour to fraction(<i>f</i>)	$4+f/2$
month to day	3	minute to minute	2
month to hour	4	minute to second	3
month to minute	5	minute to fraction(<i>f</i>)	$3+f/2$
month to second	6	second to second	2
month to fraction(<i>f</i>)	$6+f/2$	second to fraction(<i>f</i>)	$2+f/2$
day to day	2	fraction to fraction(<i>f</i>)	$1+f/2$

* When *f* is odd, round the size to the next full byte.

Figure 9-3 All possible precisions of the DATETIME data type, with their sizes in bytes

Durations: INTERVAL

The INTERVAL data type stores a duration, that is, a length of time. The difference between two DATETIME values is an INTERVAL, which represents the span of time that separates them. The following examples might help to clarify the differences:

- An employee began working on January 21, 1993 (either a DATE or a DATETIME).
- She has worked for 254 days (an INTERVAL, the difference between the TODAY function and the starting DATE or DATETIME).
- She begins work each day at 0900 hours (a DATETIME).
- She works 8 hours (an INTERVAL) with 45 minutes for lunch (another INTERVAL).
- Her quitting time is 1745 hours (the sum of the DATETIME when she begins work and the two INTERVALS).

Like DATETIME, INTERVAL is a family of types with different precisions. An INTERVAL can represent a count of years and months; or it can represent a count of days, hours, minutes, seconds, or fractions of seconds; 18 possible precisions in all. The size of an INTERVAL value ranges from 2 to 12 bytes, depending on the formulas shown in Figure 9-4.

Precision	Size*	Precision	Size*
year(p) to year	$1+p/2$	hour(p) to minute	$2+p/2$
year(p) to month	$2+p/2$	hour(p) to second	$3+p/2$
month(p) to month	$1+p/2$	hour(p) to fraction(f)	$4+(p+f)/2$
day(p) to day	$1+p/2$	minute(p) to minute	$1+p/2$
day(p) to hour	$2+p/2$	minute(p) to second	$2+p/2$
day(p) to minute	$3+p/2$	minute(p) to fraction(f)	$3+(p+f)/2$
day(p) to second	$4+p/2$	second(p) to second	$1+p/2$
day(p) to fraction(f)	$5+(p+f)/2$	second(p) to fraction(f)	$2+(p+f)/2$
hour(p) to hour	$1+p/2$	fraction to fraction(f)	$1+f/2$

* Round a fractional size to the next full byte.

Figure 9-4 *All possible precisions of the INTERVAL data type, with their sizes in bytes*

INTERVAL values can be negative as well as positive. You can add or subtract them, and you can scale them by multiplying or dividing by a number. This is not true of either DATE or DATETIME. It is reasonable to ask, “What is one-half the number of days until April 23?” but it is not reasonable to ask, “What is one-half of April 23?”

Character Types

All Informix database servers support the CHAR(*n*) data type. When NLS has been enabled, the Informix database servers support the NCHAR(*n*) data type. **INFORMIX-OnLine Dynamic Server** also supports other special use character data types.

Character Data: CHAR(*n*) and NCHAR(*n*)

The CHAR(*n*) data type contains a sequence of *n* ASCII characters. The length *n* ranges from 1 to 32,767. (If you are using the **INFORMIX-SE** database server, the maximum length is 32,511.) In NLS mode, the NCHAR(*n*) data type contains a sequence of *n* native characters with the same length range as CHAR.

Whenever a CHAR(*n*) or NCHAR(*n*) value is retrieved or stored, exactly *n* bytes are transferred. If an inserted value is short, it is extended with spaces to make up *n* bytes. A value longer than *n* is truncated. No provision exists for variable-length data in this format.

A CHAR(*n*) or NCHAR(*n*) value can include tabs and spaces but normally contains no other nonprinting characters. When rows are inserted using INSERT or UPDATE, or when rows are loaded with a utility program, no means exists for entering nonprintable characters. However, when rows are created by a program using embedded SQL, the program can insert any character except the null (binary zero) character. It is not a good idea to store nonprintable characters in a character column because standard programs and utilities do not expect them.

The advantage of the CHAR(*n*) or NCHAR(*n*) data type is that it is available on all database servers. The only disadvantage of CHAR(*n*) or NCHAR(*n*) is that its length is fixed. When the length of data values varies widely from row to row, space is wasted. NCHAR(*n*) is available on all database servers when NLS has been enabled.

Variable-Length Strings: VARCHAR(*m,r*) and NVARCHAR (*m,r*)

Often the items in a character column have different lengths; that is, many have an average length and only a few have the maximum length. The VARCHAR(*m,r*) data type (or NVARCHAR(*m, r*) when NLS has been enabled) is designed to save disk space when storing such data. A column defined as VARCHAR(*m,r*) or NVARCHAR(*m, r*) is used just like one defined as CHAR(*n*). When you define a VARCHAR(*m,r*) or NVARCHAR(*m, r*) column, you specify *m* as the *maximum* length of a data item. Only the actual contents of each item are stored on disk, with a 1-byte length field. The limit on *m* is 255; that is, a VARCHAR(*m,r*) or NVARCHAR(*m, r*) value can store as many as 255 characters. If the column is indexed, the limit on *m* is 254.

Forcing the Format of a DATETIME or INTERVAL Value

The database server always displays the components of an INTERVAL or DATETIME value in the order *year-month-day hour:minute:second.fraction*. It does not refer to the date format defined to the operating system, as it does when formatting a DATE.

You can write a SELECT statement that displays the date part of a DATETIME value in the system-defined format. The trick is to isolate the component fields using the EXTEND function and pass them through the MDY() function, which converts them to a DATE. The following code is a partial example:

```
SELECT ... MDY (
    EXTEND (DATE_RECEIVED, MONTH TO
    MONTH),
    EXTEND (DATE_RECEIVED, DAY TO DAY),
    EXTEND (DATE_RECEIVED, YEAR TO YEAR))
FROM RECEIPTS ...
```

When you are designing a report using **INFORMIX-4GL** or **INFORMIX-SQL**, you have the greater flexibility of the PRINT statement. Select each component of a DATETIME or INTERVAL value as an expression using EXTEND. Give each expression an alias for convenience.

```
SELECT ...
    EXTEND (START_TIME, HOUR TO HOUR) H,
    EXTEND (START_TIME, MINUTE TO MINUTE) M, ...
```

Then, in the report, combine the components in a PRINT expression with the desired punctuation.:

```
PRINT 'Start work at ', H USING '&&', M USING '&&', 'hours.'
```

```
Start work at 0800 hours.
```

The second parameter, *r*, is an optional *reserve* length that sets a lower limit on the length of an item as stored on disk. When an item is shorter than *r*, *r* bytes are nevertheless allocated to hold it. The purpose is to save time when rows are updated. (See “Variable-Length Execution Time” on page 9-17.)

The advantages of the `VARCHAR(m,r)` or `NVARCHAR(m,r)` data type over the `CHAR(n)` or `NCHAR(m,r)` type are found in the following list:

- It conserves disk space when the lengths of data items vary widely or when only a few items are longer than average.
- Queries on the more compact tables can be faster.

The following list describes its disadvantages:

- It does not allow lengths that exceed 255 characters.
- Updates of a table can be slower in some circumstances.
- It is not available with all Informix database servers.

Variable-Length Execution Time

When the `VARCHAR(m,r)` or `NVARCHAR(m,r)` data type is used, the rows of a table have varying lengths instead of fixed lengths. This has mixed effects on the speed of database operations.

Because more rows fit in a disk page, the database server can search the table using fewer disk operations than if the rows were of fixed length. As a result, queries can execute more quickly. Insert and delete operations can be a little quicker for the same reason.

When you update a row, the amount of work the database server must do depends on the length of the new row as compared with the old one. If the new row is the same size or shorter, execution time is not significantly different than it is with fixed-length rows. But if the new row is longer than the old one, the database server might have to perform several times as many disk operations. Thus, updates of a table that uses `VARCHAR(m,r)` or `NVARCHAR(m,r)` data can sometimes be slower than updates of a fixed-length field.

You can mitigate this effect by specifying *r* as a size that covers a high proportion of the data items. Then, most rows use the reserve length; only a little space is wasted in padding, and updates are only slow when a normal value is replaced with one of the rare, longer ones.

Large Character Objects: TEXT

The `TEXT` data type stores a block of text. It is designed to store self-contained documents: business forms, program source or data files, or memos.

Although you can store any data in a `TEXT` item, Informix tools expect that a `TEXT` item is printable, so this data type should be restricted to printable ASCII text.

How Blobs Are Used

Collectively, columns of type TEXT and BYTE are called *binary large objects*, or blobs. The database server simply stores and retrieves them. Normally, blob values are fetched and stored by programs written using **INFORMIX-4GL** or a language that supports embedded SQL, such as **INFORMIX-ESQL/C**. In such a program you can fetch, insert, or update a blob value in a manner similar to the way you read or write a sequential file.

In any SQL statement, interactive or programmed, a blob column *cannot* be used in the following ways:

- In arithmetic or Boolean expressions
- In a GROUP BY or ORDER BY clause
- In a UNIQUE test
- For indexing, either by itself or as part of a composite index

In a SELECT statement entered interactively, or in a form or report, a blob can only perform the following functions:

- Be selected by name, optionally with a subscript to extract part of it
- Have its length returned by selecting LENGTH(*column*)
- Be tested with the IS [NOT] NULL predicate

In an interactive INSERT statement, you can use the VALUES clause to insert a blob value, but the only value you can give that column is null. However, you can use the SELECT form of the INSERT statement to copy a blob value from another table.

In an interactive UPDATE statement, you can update a blob column to null or to a subquery returning a blob column.

TEXT values are not stored with the rows of which they are a part. They are allocated in whole disk pages, usually areas away from rows. (See “Locating Blob Data” on page 10-18.)

The advantage of the TEXT data type over CHAR(*n*) and VARCHAR(*m,r*) is that the size of a TEXT data item has no limit except the capacity of disk storage to hold it. The disadvantages of the TEXT data type are found in the following list:

- It is allocated in whole disk pages; a short item wastes space.
- Restrictions apply on how you can use a TEXT column in an SQL statement. (See “How Blobs Are Used” on page 9-18.)

- It is not available with all Informix database servers.

You can display TEXT values in reports generated with **INFORMIX-4GL** programs or the **ACE** report writer. You can display TEXT values on a screen and edit them using screen forms generated with **INFORMIX-4GL** programs or with the **PERFORM** screen-form processor.

Binary Objects: BYTE

The BYTE data type is designed to hold any data a program can generate: graphic images, program object files, and documents saved by any word processor or spreadsheet. The database server permits any kind of data of any length in a BYTE column.

As with TEXT, BYTE data items are stored in whole disk pages in separate disk areas from normal row data.

The advantage of the BYTE data type, as opposed to TEXT or CHAR(*n*), is that it accepts any data. Its disadvantages are the same as those of the TEXT data type.

Changing the Data Type

You can use the ALTER TABLE statement to change the data type assigned to a column after the table is built. Although this is sometimes necessary, you should avoid doing it for the following reasons:

- To change a data type, the database server must copy and rebuild the table. For large tables, that can take a lot of time and disk space.
- Some data-type changes can cause a loss of information. For example, when you change a column from a longer to a shorter character type, long values are truncated; when you change to a less-precise numeric type, low-order digits are truncated.
- Existing programs, forms, reports, and stored queries also might have to be changed.

Default Values

A default value is the value inserted into a column when an explicit value is not specified in an INSERT statement. A default value can be a literal character string defined by you or one of the following SQL null, constant expressions:

- USER
- CURRENT

- TODAY
- DBSERVERNAME

Not all columns need default values, but as you work with your data model you may discover instances where the use of a default value saves data-entry time or prevents data-entry error. For example, the telephone-directory model has a State column. While looking at the data for this column, you discover that more than 50 percent of the addresses list California as the state. To save time, you specify the string “CA” as the default value for the State column.

Check Constraints

Check constraints specify a condition or requirement on a data value before data can be assigned to a column during an INSERT or UPDATE statement. If a row evaluates to *false* for any of the check constraints defined on a table during an insert or update, the database server returns an error. You define a constraint using the CREATE TABLE or ALTER TABLE statements. For example, the following requirement constrains the values of an integer domain to a certain range:

```
Customer_Number >= 50000 AND Customer_Number <= 99999
```

You can express constraints on character-based domains this way using the MATCHES predicate and the regular-expression syntax it supports. For example, the following constraint restricts a Telephone domain to the form of a U.S. local telephone number:

```
vce_num MATCHES '[2-9][2-9][0-9]-[0-9][0-9][0-9][0-9]'
```

For additional information about check constraints, see the CREATE TABLE and ALTER TABLE statements in the *Informix Guide to SQL: Syntax*.

Creating the Database

Now you are ready to create the data model as tables in a database. You do this with the CREATE DATABASE, CREATE TABLE, and CREATE INDEX statements. The syntax of these statements is shown in detail in the *Informix Guide to SQL: Syntax*. This section discusses the use of CREATE DATABASE and CREATE TABLE in implementing a data model. The use of CREATE INDEX is covered in the next chapter.

Keep in mind that the telephone-directory data model is used for illustrative purposes only. For the sake of the example, we translate it into SQL statements.

You might have to create the same database model more than once. However, the statements that create the model can be stored and executed automatically. See “Using Command Scripts” on page 9-26 for more information about using this topic.

When the tables exist, you must populate them with rows of data. You can do this manually, with a utility program, or with custom programming.

Using CREATE DATABASE

A database is a container that holds all the parts that go into a data model. These parts include not only the tables but also views, indexes, synonyms, and other objects associated with the database. You must create a database before you can create anything else. Besides creating the database, the CREATE DATABASE statement establishes the kind of transaction logging to be used. Transaction logging is discussed in Chapter 4, “Modifying Data.”

The **INFORMIX-OnLine Dynamic Server** database server differs from other database servers in the way that it creates databases and tables. **OnLine** is discussed first.

Using CREATE DATABASE with INFORMIX-OnLine Dynamic Server

When the **OnLine** database server creates a database, it sets up records that show the existence of the database and its mode of logging. It manages disk space directly, so these records are not visible to operating system commands.

Avoiding Name Conflicts

Normally, only one copy of **OnLine** is running on a machine, and it manages the databases that belong to all users of that machine. It keeps only one list of database names. The name of your database must be different from that of any other database managed by that database server. (It is possible to run more than one copy of the database server. This is sometimes done, for example, to create a safe environment for testing apart from the operational data. In that case, you must be sure you are using the correct database server when you create the database, and again when you access it later.)

Selecting a Dbspace

OnLine offers you the option of creating the database in a particular *dbspace*. A *dbspace* is a named area of disk storage. Ask your **OnLine** administrator whether you should use a particular *dbspace*. The administrator can put a database in a *dbspace* to isolate it from other databases or to locate it on a particular disk device. (Chapter 10, “Tuning Your Data Model” discusses *dbspaces* and their relationship to disk devices.)

Some *dbspaces* are *mirrored* (duplicated on two different disk devices for high reliability); your database can be put in a mirrored *dbspace* if its contents are of exceptional importance.

Choosing the Type of Logging

OnLine offers the following four choices for transaction logging:

- No logging at all. This choice is not recommended; if the database is lost to a hardware failure, all data alterations since the last archive are lost.

```
CREATE DATABASE db_with_no_log
```

When you do not choose logging, `BEGIN WORK` and other SQL statements related to transaction processing are not permitted in the database. This affects the logic of programs that use the database.

- Regular (unbuffered) logging. This is the best choice for most databases. In the event of a failure, only uncommitted transactions are lost.

```
CREATE DATABASE a_logged_db WITH LOG
```

- Buffered logging. If the database is lost, a few or possibly none of the most recent alterations are lost. In return for this small risk, performance during alterations is improved slightly.

```
CREATE DATABASE buf_log_db WITH BUFFERED LOG
```

Buffered logging is best for databases that are updated frequently (so that speed of updating is important), but the updates can be re-created from other data in the event of a crash. You can use the `SET LOG` statement to alternate between buffered and regular logging.

- ANSI-compliant logging. This is the same as regular logging, but the ANSI rules for transaction processing are also enforced. (See the discussion of ANSI SQL in Chapter 1 of the *Informix Guide to SQL: Reference*.)

```
CREATE DATABASE std_rules_db WITH LOG MODE ANSI
```

The design of ANSI SQL prohibits the use of buffered logging.

The **OnLine** administrator can turn transaction logging on and off later. For example, it can be turned off before inserting a large number of new rows.

Using CREATE DATABASE with Other Informix Database Servers

Other Informix database servers create a database as a set of one or more files managed by the operating system. For example, under the UNIX operating system, a database is a small group of files in a directory whose name is the database name. (See the manual for your database server for details on how it uses files.) This means that the rules for database names are the same as the rules the operating system has for file names.

Choosing the Type of Logging

Other database servers offer the following three choices of logging:

- No logging at all. This choice is not recommended; if the database is lost to a hardware failure, all data alterations since the last archive are lost.

```
CREATE DATABASE not_logged_db
```

When you do not choose logging, BEGIN WORK and other SQL statements related to transaction processing are not permitted in the database. This affects the logic of programs that use the database.

- Regular logging. This is the best choice for most databases. If the database is lost, only the alteration in progress at the time of failure is lost.

```
CREATE DATABASE a_logged_db WITH LOG IN 'a-log-file'
```

You must specify a file to contain the transaction log. (The form of the file name depends on the rules of your operating system.) This file grows whenever the database is altered. Whenever the database files are archived, you should set the log file back to an empty condition so it reflects only transactions following the latest archive.

- ANSI-compliant logging. This is the same as regular logging, but the ANSI rules for transaction processing are also enabled. (See the discussion of ANSI-compliant databases in Chapter 1 of the *Informix Guide to SQL: Reference*.)

```
CREATE DATABASE std_rules_db WITH LOG IN 'a-log-file' MODE
ANSI
```

You can add a transaction log to a nonlogged database later using the START DATABASE statement.

Using CREATE TABLE

Use the CREATE TABLE statement to create each table you designed in the data model. This statement has a complicated form, but it is basically a list of the columns of the table. For each column, you supply the following information:

- The name of the column
- The data type (from the domain list you made)
- If the column (or columns) is a primary key, the constraint PRIMARY KEY
- If the column (or columns) is a foreign key, the constraint FOREIGN KEY
- If the column is not a primary key and should not allow nulls, the constraint NOT NULL
- If the column is not a primary key and should not allow duplicates, the constraint UNIQUE
- If the column has a default value, the constraint DEFAULT
- If the column has a check constraint, the constraint CHECK

In short, the CREATE TABLE statement is an image in words of the table as you drew it in the data model diagram. Figure 9-5 shows the statements for the telephone-directory model whose diagram is shown in Figure 8-16 on page 8-31.

```
CREATE TABLE name
(
    rec_num SERIAL PRIMARY KEY,
    lname CHAR(20),
    fname CHAR(20),
    bdate DATE,
    anniv DATE,
    email VARCHAR(25)
);

CREATE TABLE child
(
    child CHAR(20),
    rec_num INT,
    FOREIGN KEY (rec_num) REFERENCES name (rec_num)
);

CREATE TABLE address
(
    id_num SERIAL PRIMARY KEY,
    rec_num INT,
    street VARCHAR (50,20),
    city VARCHAR (40,10),
    state CHAR(5) DEFAULT 'CA',
    zipcode CHAR(10),
    FOREIGN KEY (rec_num) REFERENCES name (rec_num)
);

CREATE TABLE voice
(
    vce_num CHAR(13) PRIMARY KEY,
    vce_type CHAR(10),
    rec_num INT,
    FOREIGN KEY (rec_num) REFERENCES name (rec_num)
);

CREATE TABLE fax
(
    fax_num CHAR(13),
    operfrom DATETIME HOUR TO MINUTE,
    opertill DATETIME HOUR TO MINUTE,
    PRIMARY KEY (fax_num)
);

CREATE TABLE faxname
(
    fax_num CHAR(13),
    rec_num INT,
    PRIMARY KEY (fax_num, rec_num),
    FOREIGN KEY (fax_num) REFERENCES fax (fax_num),
    FOREIGN KEY (rec_num) REFERENCES name (rec_num)
);

CREATE TABLE modem
(
    mdm_num CHAR(13) PRIMARY KEY,
    rec_num INT,
    b_type CHAR(5),
    FOREIGN KEY (rec_num) REFERENCES name (rec_num)
);
```

Figure 9-5 *The CREATE TABLE statements for the telephone-directory data model*

Using Command Scripts

You can create the database and tables by entering the statements interactively. But what if you have to do it again or several more times?

You might have to do it again to make a production version after a test version is satisfactory. You might have to implement the same data model on several machines. To save time and reduce the chance of errors, you can put all the commands to create a database in a file and execute them automatically.

Capturing the Schema

You can write the statements to implement your model into a file yourself. However, you can also have a program do it for you. See Chapter 5 of the *Informix Guide to SQL: Reference*. It documents the **dbschema** utility, a program that examines the contents of a database and generates all the SQL statements required to re-create it. You can build the first version of your database interactively, making changes until it is exactly as you want it. Then you can use **dbschema** to generate the SQL statements necessary to duplicate it.

Executing the File

DB-Access or **INFORMIX-SQL**, the programs you use to enter SQL statements interactively, can be driven from a file of commands. The use of these products is covered in the *DB-Access User Manual* or the *INFORMIX-SQL User Guide*. You can start **DB-Access** or **INFORMIX-SQL** and have them read and execute a file of commands prepared by you or by **dbschema**.

An Example

Most Informix database server products come with a demonstration database called **stores6** (the one used for most of the examples in this book). The **stores6** database is delivered as an operating system command script that calls Informix products to build the database. You can copy this command script and use it as the basis for automating your own data model.

Populating the Tables

For your initial tests, it is easiest to populate the tables interactively by typing INSERT statements in **DB-Access** or **INFORMIX-SQL**. As shown in the following example, to insert a row into the **manufact** table of the **stores6** database in **DB-Access**, you enter the following command:

```
INSERT INTO manufact VALUES ('MKL', 'Martin', 15)
```

Or, if you are preparing an application program in **INFORMIX-4GL** or another language, you can use the program to enter rows.

Often, the initial rows of a large table can be derived from data held in tables in another database or in operating system files. You can move the data into your new database in a bulk operation. If the data is in another Informix database, you can retrieve it in several ways.

If you are using **INFORMIX-OnLine Dynamic Server**, you can simply select the data you want from the other database on another database server as part of an INSERT statement in your database. As shown in the following example, you could select information from the **items** table in the **stores6** database to insert into a new table:

```
INSERT INTO newtable
  SELECT item_num, order_num, quantity, stock_num,
         manu_code, total_price
  FROM stores6@otherserver:items
```

If you are using **INFORMIX-SE**, you can select the data you want from one database and insert it into another database, as long as the databases are on the same database server. As shown in the following example, you could select information from the **catalog** table in the **stores6** database to insert into a new table by using a temporary table:

```
CONNECT TO 'sharky/db1';

SELECT catalog_num, stock_num, manu_code
   FROM catalog
   INTO temptable;

DISCONNECT;

CONNECT TO 'sharky/db2';

SELECT * from temptable
   INTO newsetable;
```

If you want to select data from another database server in **INFORMIX-SE**, you must export the data to a file. You can use the **UNLOAD** statement in **DB-Access**, **INFORMIX-SQL**, or **INFORMIX-4GL**, or you can write a report in **ACE** or **INFORMIX-4GL** and direct the output to a file.

When the source is another kind of file or database, you must find a way to convert it into a flat ASCII file; that is, a file of printable data in which each line represents the contents of one table row.

After you have the data in a file, you can load it into a table using the **dbload** utility. Read about **dbload** in Chapter 5 of the *Informix Guide to SQL: Reference*. The **LOAD** statement in **DB-Access**, **INFORMIX-SQL**, or **INFORMIX-4GL** can also load rows from a flat ASCII file. See Chapter 1 of the *Informix Guide to SQL: Syntax* for information about the **LOAD** and **UNLOAD** statements.

Inserting hundreds or thousands of rows goes much faster if you turn off transaction logging. No point exists in logging these insertions because in the event of a failure you can easily re-create the lost work. The following list contains the steps of a large bulk-load operation:

- If any chance exists that other users are using the database, exclude them with the **DATABASE EXCLUSIVE** statement.
- If you are using **INFORMIX-OnLine Dynamic Server**, ask the administrator to turn off logging for the database.

The existing logs can be used to recover the database to its present state, while the bulk insertion can be run again to recover those rows if they are lost soon after.

- Perform the statements or run the utilities that load the tables with data.
- Archive the newly loaded database.

If you are using **INFORMIX-OnLine Dynamic Server**, either ask the administrator to perform a full or incremental archive, or use the **onunload** utility to make a binary copy of your database only.

If you are using other database servers, use operating system commands to back up the files that represent the database.

- Restore transaction logging and release the exclusive lock on the database.

You can enclose the steps of populating a database in a script of operating system commands. You can automate the **INFORMIX-OnLine Dynamic Server** administrator commands by invoking the command-line equivalents to ON-Monitor.

Summary

This chapter covered the following work, which you must do to implement a data model:

- Specify the domains, or constraints, that are used in the model, and complete the model diagram by assigning constraints to each column.
- Use interactive SQL to create the database and the tables in it.
- If you must create the database again, write the SQL statements to do so into a script of commands for the operating system.
- Populate the tables of the model, first using interactive SQL and then by bulk operations.
- Possibly write the bulk-load operation into a command script so you can repeat it easily.

You can now use and test your data model. If it contains very large tables, or if you must protect parts of it from certain classes of users, more work remains to be done. That is the subject of Chapter 10, “Tuning Your Data Model.”

Summary

Tuning Your Data Model

Chapter Overview	3
INFORMIX-OnLine Dynamic Server Disk Storage	3
Chunks and Pages	4
Dbspaces and Blobspaces	4
Disk Mirroring	5
Databases	5
Tables and Spaces	6
Exploiting Mirroring	6
Sharing Temporary Space	6
Assigning Dedicated Hardware	6
Reducing Contention Between Programs Using the Same Table	7
Tblspaces	7
Extents	8
Choosing Extent Sizes	8
Upper Limit on Extents	10
Reorganizing dbspaces and Tables	10
Calculating Table Sizes	12
Estimating Fixed-Length Rows	13
Estimating Variable-Length Rows	14
Estimating Index Pages	16
Estimating Blobpages	17
Locating Blob Data	18
Managing Indexes	19
Space Costs of Indexes	20
Time Costs of Indexes	20

Choosing Indexes	21
Join Columns	21
Selective Filter Columns in Large Tables	22
Order-By and Group-By Columns	22
Duplicate Keys Slow Index Modifications	22
Dropping Indexes	24
Clustered Indexes	25
Denormalizing	26
Shorter Rows for Faster Queries	26
Expelling Long Strings	26
Using VARCHAR Strings	26
Changing Long Strings to TEXT	27
Building a Symbol Table of Repeated Strings	27
Moving Strings to a Companion Table	28
Splitting Wide Tables	28
Division by Bulk	28
Division by Frequency of Use	28
Division by Frequency of Update	29
Costs of Companion Tables	29
Splitting Tall Tables	29
Redundant and Derived Data	30
Adding Derived Data	30
Adding Redundant Data	31
Maximizing Concurrency	32
Easing Contention	32
Rescheduling Modifications	33
Using an Update Journal	33
Isolating and Dispersing Updates	35
Splitting Tables to Isolate Volatile Columns	35
Dispersing Bottleneck Tables	35
Summary	36

Chapter Overview

The preceding chapters summarized the steps of creating and implementing a data model that is theoretically sound. Normally, the resulting database also yields adequate performance. If it does, the job is done.

However, some applications have more stringent performance requirements. Some databases contain extremely large tables, others must be usable by many programs concurrently or by programs that operate under tight requirements for response time, and others must be extremely reliable.

You can tune your database design to meet such requirements. In some cases, you might have to sacrifice theoretical correctness in the data model to get the performance you need.

Before you make any changes, however, make sure that the queries are as efficient as possible. Study Chapter 13, “Optimizing Your Queries,” to see what you can do to improve performance without changing the data model.

The remainder of this chapter contains discussions on the following topics:

- The disk storage methods used by **INFORMIX-OnLine Dynamic Server** and their implications for performance and reliability
- The methods for calculating the sizes of tables and indexes, and using those numbers to estimate query-execution time
- The benefits and costs of adding indexes
- The ways you can “denormalize” the data model, that is, spoil its theoretical purity to improve performance
- Suggestions for increasing concurrency among multiple programs

INFORMIX-OnLine Dynamic Server Disk Storage

OnLine manages disk storage directly. To the greatest extent possible, it bypasses the host operating system and works with the raw disk space. (The extent to which this is possible depends on the operating system.)

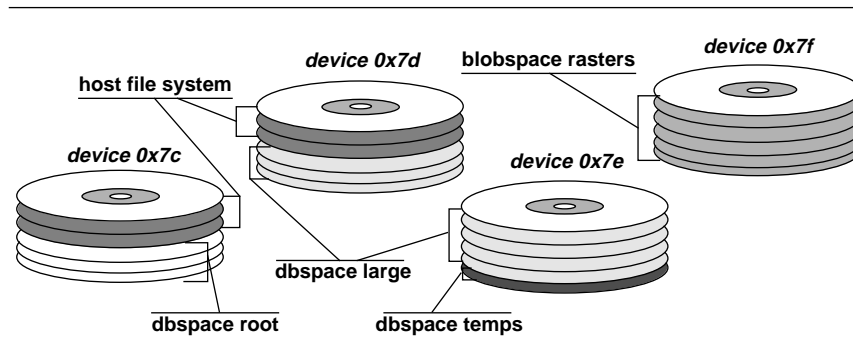


Figure 10-1 The relationship between chunks and dbspaces

Its methods of disk management are not complicated. However, several terms are used for different amounts of space in varying contexts: the *chunk*, the *dbspace* (or *blobspace*), the *page*, the *extent*, and the *tblspace*.

Chunks and Pages

The basic unit of physical storage is the *chunk*. A chunk is a unit of disk storage that is dedicated to **OnLine**. Usually it is a physical device, that is, a disk drive. However, a chunk can be part of one disk or simply a file. In any case, a chunk is a unit of space that can be identified to the operating system as being in the exclusive control of **OnLine**.

The *page* is the basic unit of disk input/output (I/O). All space in every chunk is divided into pages. All I/O is done in units of whole pages. The size of a page is the same in all chunks used for tables. It is set when an **OnLine** system is installed.

Dbspaces and Blobspaces

A *space* always comprises one or more complete chunks. Often a space contains exactly one chunk, so that the space and the chunk are identical. However, when a space comprises more than one chunk, **OnLine** makes the multiple chunks appear to be a single sequence of pages.

A space is either a *dbspace* or a *blobspace*, depending on its use. If a space contains databases and tables, it is a *dbspace*. If it is dedicated to holding blobs (the TEXT and BYTE data types), it is a *blobspace*. Chunks are assigned to a space when it is created. Chunks also can be added to a space later.

One dbspace, the *root* dbspace, is created first and always exists. It is the most important space because it holds the control information that describes all other spaces and chunks.

Parts of a single database can appear in two or more dbspaces. However, a single table is always completely contained in one dbspace.

Disk Mirroring

Individual spaces can be *mirrored*. The chunks of a mirrored space are paired with other chunks of equal size. Whenever a page is written to one of the chunks, it is also written to the mirror chunk. When one chunk suffers a hardware failure, the database server uses its mirror to continue processing without interruption. When the broken chunk is restored to service (or when a substitute chunk is assigned), **OnLine** automatically brings it back to equality with the working chunk and continues operations.

When any dbspaces are mirrored, the root dbspace should also be mirrored. Otherwise, if it is lost to hardware failure, all **OnLine** data is unusable regardless of mirrors.

If your database has extreme requirements for reliability in the face of hardware failure, you should place it in a mirrored dbspace. As indicated in “Tables and Spaces” on page 10-6, it is possible to locate individual tables in particular dbspaces. Hence, you can place some tables in mirrored spaces and other tables in normal spaces.

Databases

A database resides initially in one dbspace. It is placed there by a parameter of the CREATE DATABASE statement. The following example creates a database in the dbspace named **dev0x2d**:

```
CREATE DATABASE reliable IN dev0x2d WITH BUFFERED LOG
```

When no dbspace is specified, a database is placed in the root dbspace. When a database is in a dbspace it means only that the following conditions have been met:

- Its system catalog tables are stored in that dbspace
- That dbspace is the default location for tables that are not explicitly created in other dbspaces

Tables and Spaces

A table resides completely in one dbspace. (Its blob values can reside in separate blobspaces or on an optical storage subsystem.) If no dbspace is specified, a table resides in the dbspace where its database resides. The following partial example creates a table in the dbspace named **misctabs**:

```
CREATE TABLE taxrates (...column specifications...) IN misctabs
```

You can achieve many aims by placing a table in a specific dbspace. Some of these aims are explored in the following paragraphs.

Exploiting Mirroring

Place all the tables used by a critically important application in a mirrored dbspace. Alternatively, create the database in the mirrored dbspace and let the important tables reside there by default. Tables that are part of the database, but that are not used by the critical programs, can be located in nonmirrored spaces.

Sharing Temporary Space

When databases are large and disk space is limited, the normal dbspaces might not have enough room for those large temporary tables that can be helpful in improving query performance. However, a temporary table exists only for the life of the program that creates it—even less time if the program drops the table promptly. Set up a single dbspace for temporary tables, so it can be used repeatedly by many programs.

Assigning Dedicated Hardware

A dbspace can equal a chunk, and a chunk can equal a device. Thus, you can place a table on a disk device that is dedicated to its use. When disk drives have different performance levels, you can put the tables with the highest frequency of use on the fastest drives.

By giving a high-use table a dedicated access arm, you can reduce contention with applications that use other tables. This does not reduce contention between programs using the same table unless the table itself is spread across multiple devices, as suggested in the next section.

Reducing Contention Between Programs Using the Same Table

A dbspace can comprise multiple chunks, and each chunk can represent a different disk. This permits you to assign multiple disk-access arms to one table. Figure 10-2 shows a dbspace distributed over multiple disks.

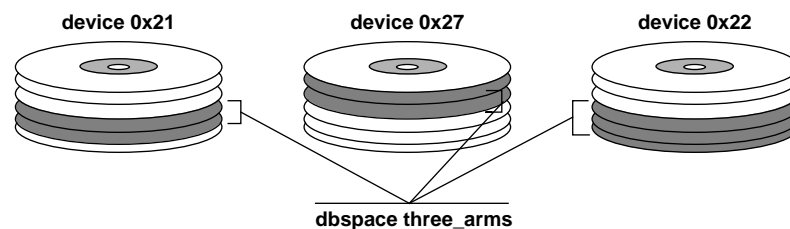


Figure 10-2 A dbspace distributed over three disks

The dbspace named **three_arms** comprises chunks on three different disks. Each chunk is approximately one-third the size of an important, heavily used table. (The third chunk is larger to allow for growth.) When multiple programs query the table, their I/O operations are distributed over the three disks, reducing contention for the use of the hardware.

In an arrangement like this, the multiple disks can act as mirrors for each other. For example, device **0x21** can be mirrored on **0x27**, **0x27** on **0x22**, and **0x22** on **0x21**. As long as any two devices are operational, the dbspace is usable. (However, if a single piece of hardware, such as a disk controller, is common to all three disks, this type of mirror arrangement is still vulnerable to the failure of that component.)

Tblspaces

The total of all disk space allocated to a table is the *tblspace* of the table. The *tblspace* includes pages allocated to data (rows of that table) and pages allocated to indexes. It also includes pages used by blob columns that are located in the *tblspace* but not pages used by blob data in a separate *blob*space or blob data stored on an optical storage subsystem. (This choice is discussed under “Locating Blob Data” on page 10-18.)

The *tblspace* is an accounting entity only; it does not correspond to any particular part of a *dbspace*. The indexes and data extents that make up a table can be scattered throughout the *dbspace*.

The **oncheck** utility (with the **-pt** option) returns information on the status of tablespaces, including the counts of pages allocated and in use for each one. You can use this information to monitor the growth of a table over time.

Extents

As you add rows to a table, **OnLine** allocates disk space to it in units called *extents*. Each extent is a block of physically contiguous pages from the dbspace. Even when the dbspace comprises more than one chunk, extents are always allocated entirely within a single chunk to remain contiguous.

Contiguity is important to performance. When the pages of data are contiguous, disk-arm motion is minimized when the database server reads the rows sequentially. The mechanism of extents is a compromise between the following competing requirements:

- Most dbspaces are shared among several tables.
- The size of some tables is not known in advance.
- Tables can grow at different times and different rates.
- All the pages of a table should be adjacent for best performance.

Because table sizes are not known, table space cannot be preallocated. Therefore, extents are added only as they are needed, but all the pages in any one extent are contiguous for better performance.

Choosing Extent Sizes

When you create a table, you can specify the size of the first extent as well as the size of the extents to be added as the table grows. You can change the size of added extents later with the ALTER TABLE statement. The following partial example creates a table with a half-megabyte initial extent and 100-kilobyte added extents:

```
CREATE TABLE big_one (...column specifications...)
  IN big_space
  EXTENT SIZE 512
  NEXT SIZE 100
```

The following example changes the next-extent size of the table to 50 kilobytes. This has no effect on extents that already exist.

```
ALTER TABLE big_one MODIFY NEXT SIZE 50
```

The next-extent sizes of the following kinds of tables are not very important to performance:

- A small table has only one extent (otherwise it would not be small). If it is heavily used, large parts of it are buffered in memory anyway.
- An infrequently used table is not important to performance no matter what size it is.
- A table that resides in a dedicated dbspace always receives new extents that are adjacent to its old extents. The size of these extents is not important because, being adjacent, they perform like one large extent.

When you assign an extent size to these kinds of tables, the only consideration is to avoid creating large numbers of extents. A large number of extents causes the database server to spend a small amount of extra time on bookkeeping. In addition, an upper limit exists on the number of extents allowed. (This is covered in the section “Upper Limit on Extents” on page 10-10.)

Next-extent sizes become important when two or more large and growing tables share a dbspace. Because the extents added to the different tables are interleaved, each new extent represents another long seek that must be made when reading sequentially. It also extends the total distance over which the disk arm must range when reading nonsequentially.

No upper limit exists on extent sizes except the size of the dbspace. When you know the final size of a table (or can confidently predict it within 25 percent), allocate all its space in the initial extent. When tables grow steadily to unknown size, assign them next-extent sizes that let them share the dbspace with a small number of extents each. The following list contains one possible approach:

- Decide on the ratio in which the dbspace is shared among the tables. For example, you might divide the dbspace among three tables in the ratio 0.4 : 0.2 : 0.3 (reserving 10 percent for small tables and overhead).
- Give each table one-fourth of its share of the dbspace as its initial extent.
- Assign each table one-eighth of its share as its next-extent size.

Monitor the growth of the tables regularly using **oncheck**.

What happens if, as the dbspace fills up, you do not have enough contiguous space to create an extent of the size you specified? In that case, **OnLine** allocates the largest contiguous extent that it can.

Upper Limit on Extents

No table should be allowed to acquire a large number of extents. But it can happen, and an upper limit exists on the number of extents allowed. Trying to add an extent after the limit is reached causes **ISAM** error -136 (No more extents) to follow an INSERT request.

The upper limit depends on the page size and the table definition. To learn the upper limit on extents for a particular table, calculate a series of values as shown in the following list:

- pagesize* = the size of a page (reported by **onstat -c** under the head BUFFSIZE)
- colspace* = $4 \times$ the number of columns in the table
- ixspace* = $12 \times$ the number of indexes on the table
- ixparts* = $4 \times$ the number of columns named in each index
- extspace* = $pagesize - (colspace + ixspace + ixparts + 84)$

The table can have no more than *extspace*/8 extents. When the page size is 2,048, the **customer** table from the demonstration database can have no more than 234 extents, as calculated in Figure 10-3.

To help ensure that the limit is not exceeded, **OnLine** checks the number of extents each time it creates a new one. If the extent being created is the 16th or a multiple of 16, the next-extent size for the table is automatically doubled.

Variable	Value	Basis
<i>pagesize</i>	2,048	output of onstat
<i>colspace</i>	40	ten columns
<i>ixspace</i>	36	three indexes
<i>ixparts</i>	12	one column in each index
<i>extspace</i>	1,876	$2,048 - (40 + 36 + 12 + 84)$
<i>limit</i>	234	$extspace / 8$

Figure 10-3 Values used to determine upper limit on extents

Reorganizing dbspaces and Tables

Query performance can suffer after many extents are added to a table. This can only be the case when two or more large, growing tables share a dbspace. When tables grow at the same time, their new extents and index pages are interleaved, creating large gaps between the extents of any one table. This situation is shown in Figure 10-4.



Figure 10-4 *A dbspace with interleaved tables*

Figure 10-4 also shows gaps of unused space, perhaps created by tables that existed when extents were allocated but that have since been dropped.

Disorganization such as that depicted in Figure 10-4 hurts performance in two ways. During sequential access to any table, several long seeks must be made. For nonsequential access to any table, the disk might have to seek across the entire width of the dbspace. It is possible to rebuild a dbspace so that tables are compact once again, as shown in Figure 10-5. The relative order of the reorganized tables within the dbspace is not important; all that matters is that the pages of each reorganized table are together. When reading a table sequentially, no long seeks occur. When reading a table nonsequentially, the disk arm ranges only over the space occupied by that table.



Figure 10-5 *A dbspace reorganized to make tables compact*

Use the following steps to reorganize a dbspace:

1. Copy the tables in the dbspace to tape individually, using the **onunload** utility.
2. Drop all the tables in the dbspace.
3. Re-create the tables using the **onload** utility.

The **onload** utility re-creates the tables with the identical properties they had before, including the same extent sizes. When a new extent is created adjacent to the previous extent, the two are treated as a single extent.

You can also unload a table using the UNLOAD statement in **DB-Access**, **INFORMIX-SQL**, or **INFORMIX-4GL**, and reload the table using the companion LOAD statement or the **dbload** utility. However, these operations convert the table into character form, while **onload** and **onunload** work with binary copies of disk pages.

You can reorganize a single table in two more ways. If you use the ALTER TABLE statement to add or drop a column or to change the data type of a column, the table is copied and reconstructed. If you create a clustered index or alter an index to cluster, the table is sorted and rewritten. (See “Clustered Indexes” on page 10-25.) In both cases, the table is written on other areas of the dbspace. However, if other tables are in the dbspace, no guarantee exists that all the new extents are adjacent.

Calculating Table Sizes

This section discusses methods for calculating the approximate sizes of tables and indexes in disk pages. As in the previous sections, this information applies only to **INFORMIX-OnLine Dynamic Server**. Estimates of this kind are useful when you are planning the layout of a database on disk.

These calculations aim to estimate the number of disk pages that are used to hold different parts of a table. If the table already exists, or if you can build a demonstration table of realistic size using simulated data, you do not have to make estimates. You can run **oncheck** and obtain exact numbers.

You can calculate a series of values to produce the estimates. You need the following values in all estimates. Let the values be as follows:

estrows = the number of rows you estimate the table has when it reaches its normal size.

pagesize = the size of a page (reported by **onstat -c** under the head BUFFSIZE).

pageuse = *pagesize* - 28 (the space available for data on a disk page).

In the calculations in this chapter, **floor(x)** means the largest integer smaller than *x*, and **ceiling(x)** means the smallest integer larger than *x*.

Estimating Fixed-Length Rows

When a table contains no VARCHAR columns, its rows all have the same size. Calculate the following additional variables. Let the values be as shown in the following list:

rowsize = the sum of the sizes of all columns in the table, plus 4 bytes. Sizes of different data types are discussed in Chapter 9, “Implementing Your Data Model.” Treat TEXT and BYTE columns as having size 56.

homerow = if (*rowsize* ≤ *pageuse*) then *rowsize*
else 4 + remainder (*rowsize*/*pageuse*)

If a row is larger than a page, as many full pages as possible are removed from the end of the row and stored in separate expansion pages. Let *homerow* be the size of the leading part, which is kept on the home page. The value *homerow* is simply *rowsize* if a row fits a page.

overpage = if (*rowsize* ≤ *pageuse*) then 0
else floor (*rowsize*/*pageuse*)

Let *overpage* be the number of expansion pages needed for each row; it is zero when a row fits on a page.

datrows = min (255, floor [*pageuse*/*homerow*])

Let *datrows* be the number of rows (or leading parts of rows) that fits on one page. An upper limit exists of 255 rows per page, even when rows are very short.

datpages = ceiling (*estrows*/*datrows*)

expages = *estrows* × *overpage*

The table requires approximately *datpages* + *expages* disk pages for rows. (Additional pages for TEXT and BYTE columns are estimated in “Estimating Blobpages” on page 10-17.) An estimate for the **customer** table is shown in Figure 10-6.

Variable	Estimate	Basis of Estimate
<i>estrows</i>	1,500	company business plan
<i>pagesize</i>	2,048	output of onstat
<i>pageuse</i>	2,020	<i>pagesize</i> - 28
<i>rowsize</i>	138	sum of 4+ column
		customer_num data type size
		customer_num SERIAL 4
		fname CHAR(15) 15
		lname CHAR(15) 15
		company CHAR(20) 20
		address1 CHAR(20) 20
		address2 CHAR(20) 20
		city CHAR(15) 15
		state CHAR(2) 2
		zipcode CHAR(5) 5
		phone CHAR(18) 18
<i>homerow</i>	134	$rowsize \leq pageuse$
<i>overpage</i>	0	$rowsize \leq pageuse$
<i>datrows</i>	14	$\text{floor}(pageuse / homerow)$
<i>datpages</i>	108	$\text{ceiling}(estrows / datrows)$
<i>expages</i>	0	$estrows \times overpage$

Figure 10-6 Estimating the size of the customer table

Estimating Variable-Length Rows

When a table contains one or more VARCHAR columns (or NVARCHAR columns, when NLS has been enabled), its rows have varying lengths. This introduces uncertainty. You must form an estimate of the typical size of each VARCHAR column, based on your understanding of the data.

When INFORMIX-OnLine Dynamic Server allocates space to rows of varying size, it considers a page to be full when no room exists for an additional row of the *maximum* size. This can limit the allocation of rows to pages. Calculate the following variables:

maxrow = the sum of the maximum sizes of all columns in the table, plus 4 bytes. Treat TEXT and BYTE columns as having size 56.

typro = the sum of the estimated typical sizes of all columns in the table, plus 4 bytes.

homerow = if (*typro* ≤ *pageuse*) then *typro*
 else 4 + remainder (*typro* / *pageuse*)

Let *homerow* be the amount of a typical row that fits on the home page; it is *typro* if one page is sufficient.

$overpage =$ if ($typrow \leq pageuse$) then 0
else floor ($typrow / pageuse$)

Let $overpage$ be the number of expansion pages needed for the typical row; it is zero when a typical row fits on a page.

$homemax =$ if ($maxrow \leq pageuse$) then $maxrow$
else 4 + remainder ($maxrow / pageuse$)

Let $homemax$ be the reserve requirement of the database server.

$datrows =$ floor ($(pageuse - homemax) / typrow$)

Let $datrows$ be the number of typical rows that fits on a page before the database server decides the page is full.

$datpages =$ ceiling ($estrows / datrows$)

$expages =$ $estrows \times overpage$

The table requires approximately $datpages + expages$ disk pages for rows. (Additional pages for TEXT and BYTE columns are estimated in “Estimating Blobpages” on page 10-17.) An estimate for the **catalog** table is shown in Figure 10-6.

Variable	Estimate	Basis of Estimate																												
<i>estrows</i>	5,000	company business plan.																												
<i>pagesize</i>	2,048	output of onstat																												
<i>pageuse</i>	2,020	<i>pagesize</i> - 28																												
<i>maxrow</i>	381	sum of 4+:																												
		<table><tr><th>column</th><th>data type</th><th>max size</th><th>typical</th></tr><tr><td>catalog_num</td><td>SERIAL</td><td>4</td><td>4</td></tr><tr><td>stock_num</td><td>SMALLINT</td><td>2</td><td>2</td></tr><tr><td>manu_code</td><td>CHAR(3)</td><td>3</td><td>3</td></tr><tr><td>cat_descr</td><td>TEXT</td><td>56</td><td>56</td></tr><tr><td>cat_picture</td><td>BYTE</td><td>56</td><td>56</td></tr><tr><td>cat_adv</td><td>VARCHAR(255,65)</td><td>256</td><td>66</td></tr></table>	column	data type	max size	typical	catalog_num	SERIAL	4	4	stock_num	SMALLINT	2	2	manu_code	CHAR(3)	3	3	cat_descr	TEXT	56	56	cat_picture	BYTE	56	56	cat_adv	VARCHAR(255,65)	256	66
column	data type	max size	typical																											
catalog_num	SERIAL	4	4																											
stock_num	SMALLINT	2	2																											
manu_code	CHAR(3)	3	3																											
cat_descr	TEXT	56	56																											
cat_picture	BYTE	56	56																											
cat_adv	VARCHAR(255,65)	256	66																											
<i>typrow</i>	191																													
<i>homerow</i>	191	<i>typrow</i> ≤ <i>pageuse</i>																												
<i>overpage</i>	0	<i>typrow</i> ≤ <i>pageuse</i>																												
<i>homemax</i>	381	<i>maxrow</i> ≤ <i>pageuse</i>																												
<i>datrows</i>	8	floor (<i>(pageuse</i> - <i>homemax</i>) / <i>typrow</i>)																												
<i>datpages</i>	625	ceiling (<i>estrows</i> / <i>datrows</i>)																												
<i>expages</i>	0	<i>estrows</i> × <i>overpage</i>																												

Figure 10-7 Estimating the size of the catalog table

Estimating Index Pages

The `tblspace` includes index pages as well as pages of data, and index pages can be a significant fraction of the total.

An index entry consists of a *key* and a *pointer*. The key is a copy of the indexed column or columns from one row of data. The pointer is a 4-byte value used to locate the row containing that key value.

A unique index contains one such entry for every row in the table. It also contains some additional pages of pointers that create the B+ tree structure. (Index structure is described in “The Cost of Network Access” on page 13-20.) When an index allows duplicates, it contains fewer keys than pointers; that is, one key can be followed by a list of pointers.

If this were the whole story on indexes, it would be simple to estimate their space requirements. However, the index leaf pages use a technique called *key compression*, which allows the database server to put more keys on a page than would otherwise fit. The space saved by key compression varies with the content of the keys, from page to page and index to index.

To estimate the size of a compact index (disregarding space saved by key compression), let the values be as shown in the following list:

<i>keysize</i> =	the total width of the indexed column or columns.
<i>pctuniq</i> =	the number of unique entries you expect in this index, divided by <i>estrows</i> .
	For a unique index, or one with only occasional duplicate values, use 1.0. When duplicates are present in significant numbers, let <i>pctuniq</i> be a fraction less than 1.0.
<i>entsize</i> =	$\text{keysize} + (5/\text{pctuniq})$
<i>pagents</i> =	$\text{floor}(\text{pageuse}/\text{entsize})$
	Approximately <i>pagents</i> entries occur on each index page.
<i>leaves</i> =	$\text{ceiling}(\text{estrows}/\text{pagents})$
	Approximately <i>leaves</i> leaf pages occur in the index.
<i>branches</i> =	$\text{ceiling}(\text{leaves}/\text{pagents})$
	There are approximately <i>branches</i> nonleaf pages.

The index contains approximately *leaves* + *branches* pages when it is compact. Estimates for two indexes in the **customer** table are carried out in Figure 10-8 on page 10-17.

As rows are deleted and new ones inserted, the index can become sparse; that is, the leaf pages might no longer be full of entries. On the other hand, if key compression is effective, the index might be smaller. The method given here should yield a conservative estimate for most indexes. If index space is important, build a large test index using real data and check its size with the **oncheck** utility.

Variable	Estimate	Basis for Estimate
<i>estrows</i>	1,500	company business plan
<i>pagesize</i>	2,048	output of onstat
<i>pageuse</i>	2,016	<i>pagesize</i>
<i>keysize</i>	4	customer_num is 4 bytes
<i>pctuniq</i>	1.0	unique index
<i>entsize</i>	12	$4 + (5/1)$
<i>pagents</i>	168	floor $(2,016/12)$
<i>leaves</i>	9	ceiling $(1,500/168)$
<i>branches</i>	1	ceiling $(9/168)$
<i>keysize</i>	2	state is CHAR(2)
<i>pctuniq</i>	0.033	50 states among 1,500 rows
<i>entsize</i>	8.066	$2 + (5/0.033)$
<i>pagents</i>	249	floor $(2,016/8.066)$
<i>leaves</i>	7	ceiling $(1,500/249)$
<i>branches</i>	1	ceiling $(7/168)$

Figure 10-8 Estimating the size of two indexes from the customer table

Estimating Blobpages

BYTE and TEXT data items that are stored on magnetic disk are stored in separate pages, either interspersed among the row and index pages in the *tblspace* or in a separate *blobspace*. (This discussion does not apply to blob data stored on optical media.) Each blob data item occupies a whole number of pages. For each blob column, let the values be as shown in the following list:

- typage* = the number of whole pages required to store an item of typical size.
- nnpart* = the fraction of rows in which this column has a non-null value.
- totpage* = $typage \times estrows \times nnpart$.

The values for the column occupy approximately *totpage* pages. In the **catalog** table (see Figure 10-6 on page 10-14) two blobpages occur. Their estimates can be imagined as described in the following paragraphs:

For the **cat_descr** column, the text of a description is, at most, a double-spaced page, which is 250 words or approximately 1,500 characters, so *typage* = 1. A description exists for every entry, so *nnpart* = 1.0. Thus, *totpage* = $1 \times 5,000 \times 1.0 = 5,000$ pages of data.

The **cat_picture** column contains line art in a computer-readable form. Examining several of these pictures reveals that they vary widely in size but a typical file contains about 75,000 bytes of data. Thus *typage* = 38. The marketing department estimates that they want to store a picture with one entry in four: *nnpart* = 0.25. Therefore *totpage* = $38 \times 5,000 \times 0.25 = 47,500$ pages of data.

After a table is built and loaded, you can check the use of blobpages with the **oncheck** utility **-ptT** option.

Locating Blob Data

When you create a BYTE or TEXT column on magnetic disk, you have the option of locating the data of the column in the tblspace or in a blobspace. (You can also store blob data on optical media, although this discussion does not apply to blobs stored in this way.) In the following example, a TEXT value is located in the tblspace and a BYTE value is located in a blobspace named **rasters**:

```
CREATE TABLE examptab
(
    pic_id SERIAL,
    pic_desc TEXT IN TABLE,
    pic_raster BYTE IN rasters
)
```

A TEXT or BYTE value is always stored apart from the rows of the table. Only a 56-byte descriptor is stored with the row. However, the value itself occupies at least one disk page.

When blob values are stored in the tblspace, the pages of their data are interspersed among the pages containing rows. The result is to inflate the size of the table. The blobpages separate the pages containing rows and spread them out on the disk. When the database server reads only the rows and not the blob data, the disk arm must move farther than it would if the blobpages

were stored apart. The database server scans only the row pages on any SELECT operation that retrieves no blob column and whenever it tests rows using a filter expression.

Another consideration is that disk I/O to and from a dbspace is buffered. Pages are held in storage in case they are needed again soon; and, when pages are written, the requesting program is allowed to continue before the actual disk write takes place.

However, disk I/O to and from blobspaces is not buffered. Blobspace pages are not retained in buffers to be read again, and the requesting program is not allowed to proceed until all output to them is complete. The reason is that blobspace I/O is expected to be voluminous. If these pages are passed through the normal buffering mechanisms, they could monopolize the buffers, driving out index pages and other pages that are useful to good performance.

For best performance, then, you should locate a TEXT or BYTE column in a blobspace in either of the following circumstances:

- When single data items are larger than one or two pages each; if kept in the dbspace, their transfer dilutes the effectiveness of the page buffers.
- When the number of pages of blob data is more than half the number of pages of row data; if kept in the dbspace, the table is inflated and queries against it are slowed.

For a table that is both relatively small and nonvolatile, you can achieve the effect of a dedicated blobspace by the following means: Load the entire table with rows in which the blob columns are null. Create all indexes. The row pages and the index pages are now contiguous. Update all the rows to install the blob data. The blob pages follow the pages of row and index data in the tblspace.

Managing Indexes

An index is necessary on any column (or composition of columns) that must be unique. However, as discussed in Chapter 13, “Optimizing Your Queries,” the presence of an index can also allow the query optimizer to speed up a query. The optimizer can use an index in the following ways:

- To replace repeated sequential scans of a table with nonsequential access
- To avoid reading row data at all when processing expressions that name only indexed columns

- To avoid a sort (including building a temporary table) when executing the GROUP BY and ORDER BY clauses

As a result, an index on the right column can save thousands, tens of thousands, or in extreme cases, even millions of disk operations during a query. However, costs are associated with indexes.

Space Costs of Indexes

The first cost of an index is disk space. An estimating method is given in “Estimating Index Pages” on page 10-16. Loosely, an index contains a copy of every unique data value in the indexed columns, plus a 4-byte pointer for every row in the table. This information can add many pages to the table space requirements; it is easy to have as many index pages as row pages.

Time Costs of Indexes

The second cost of an index is time while the table is modified. The descriptions that follow assume that approximately two pages must be read to locate an index entry. That is the case when the index consists of a root page, one intermediate level, and leaf pages, and the root page is already in a buffer. The index for a very large table has two intermediate levels, so roughly three pages are read when an entry is looked up.

Presumably, one index has been used to locate the row being altered. Its index pages are found in the page buffers. However, the pages for any other indexes that need altering must be read from disk.

Under these assumptions, index maintenance adds time to different kinds of modifications as shown in the following list:

- When you delete a row from a table, its entries must be deleted from all indexes.

The entry for the deleted row must be looked up (two or three pages in) and the leaf page must be rewritten (one page out), for a total of three or four page accesses per index.

- When you insert a row, its entries must be inserted in all indexes.

The place for entry of the inserted row must be found (two or three pages in) and rewritten (one page out), for a total of three or four page accesses per index.

- When you update a row, its entries must be looked up in each index that applies to a column that was altered (two or three pages in). The leaf page must be rewritten to eliminate the old entry (one page out); then the new

column value must be located in the same index (two or three more pages in) and the row entered (one more page out).

Insertions and deletions change the number of entries on a leaf page. In virtually every *pagents* operation, some additional work must be done because a leaf page has either filled up or been emptied. However, because *pagents* is usually greater than 100, this occurs less than 1 percent of the time and can be disregarded for estimating.

In short, when a row is inserted or deleted at random, allow three to four added page I/O operations per index. When a row is updated, allow six to eight page I/O operations for each index that applies to an altered column. Bear in mind also that if a transaction is rolled back, all this work must be undone. For this reason, rolling back a transaction can take a long time.

Because the alteration of the row itself requires only two page I/O operations, it is clear that index maintenance is the most time-consuming part of data modification. One way to reduce this cost is discussed under “Dropping Indexes” on page 10-24.

Choosing Indexes

Indexes are required on columns that must be unique and are not specified as primary keys. In addition, you should add an index in the following cases:

- Columns used in joins that are not specified as foreign keys
- Columns frequently used in filter expressions
- Columns frequently used for ordering or grouping

Join Columns

As discussed in Chapter 13, “Optimizing Your Queries,” at least one column named in any join expression should have an index. If no index exists, the database server may build a temporary index or perform a sort-merge before the join and discard it afterward; that is almost always faster than performing a join by repeated sequential scans over a table.

When both columns in a join expression have indexes, the optimizer has more options when it constructs the query plan. As a general rule, you should put an index on any column that is not identified as a primary or foreign key and is used in a join expression more than occasionally.

Selective Filter Columns in Large Tables

If a column is often used to filter the rows of a large table, consider placing an index on it. The optimizer can use the index to select the desired columns, avoiding a sequential scan of the entire table. One example might be a table that contains a large mailing list. If you find that a postal-code column is often used to filter a subset of rows, you should consider putting an index on it even though it is not used in joins.

This strategy yields a net savings of time only when the selectivity of the column is high; that is, when only a small fraction of rows holds any one indexed value. Nonsequential access through an index takes several more disk I/O operations than sequential access does, so if a filter expression on the column passes more than a fourth of the rows, the database server might as well read the table sequentially. As a rule, indexing a filter column saves time in the following cases:

- The column is used in filter expressions in many queries or in slow queries.
- The column contains at least 100 unique values.
- Most column values appear in fewer than 10 percent of the rows.

Order-By and Group-By Columns

When a large quantity of rows must be ordered or grouped, the database server must put the rows in order. One way the database server does this is to select all the rows into a temporary table and sort the table. But (as discussed in Chapter 13, “Optimizing Your Queries”) if the ordering columns are indexed, the optimizer sometimes plans to read the rows in sorted order through the index, thus avoiding a final sort.

Because the keys in an index are in sorted sequence, the index really represents the result of sorting the table. By placing an index on the ordering column or columns, you can replace many sorts during queries with a single sort when the index is created.

Duplicate Keys Slow Index Modifications

When duplicate keys are permitted in an index, the entries that have any single value are grouped in a list. When the selectivity of the column is high, these lists are generally short. But when only a few unique values occur, the lists become quite long, and in fact, can cross multiple leaf pages.

For example, in an index on a column whose only values are *M* for married and *S* for single, all the index entries are contained in just two lists of duplicates. Such an index is not of much use, but at least it works for querying; the database server can read out the list of rows that have one value or the other.

When an entry must be deleted from a list of duplicates, the database server must read the whole list and rewrite some part of it. When it adds an entry, the database server puts the new row at the end of its list. Neither operation is a problem when the list is short, which is normal. But when a list fills many pages, the database server must read all the rows to find the end. When it deletes an entry, it typically must update and rewrite half the pages in the list.

Thus, an index on a column that has a small number of distinct values, in a table that has a large number of rows, can drastically reduce the speed of updating. An example is a column whose values are the names or abbreviations of states or provinces. If 50 unique values occur in a mailing list of 100,000 rows, an average of 2,000 duplicates exists per value. But real data is never so well distributed; in such a table, the more common values likely have 10,000 or more duplicates, and their lists might approach 50 pages in length.

When the database server inserts or deletes an entry in such a list, it is busy for a long time. Worse still, it has to lock all the affected index pages while it does the work, greatly reducing concurrent access to the table.

You can avoid this problem fairly simply at some cost in disk space. The trick is to know that the database server uses the leading column of a composite index in the same way as it uses an index on that column alone. So instead of creating an index on a column with few unique values, create a composite index on that column followed by one other column that has a wide distribution of values.

For example, change the index on the column whose only values are *M* and *S* into a composite index on that column and a birthdate column. You can use any second column to disperse the key values as long as its value does not change or changes at the same time as the real key. The shorter the second column the better because its values are copied into the index and expand its size.

Dropping Indexes

In some applications, the majority of table updates can be confined to a single time period. Perhaps all updates are applied overnight or on specified dates.

When this is the case, consider dropping all non-unique indexes while you perform updates and then create new indexes afterward. This can have two good effects.

First, because fewer indexes occur to update, the updating program can run faster. Often, the total time to drop the indexes, update without them, and re-create them afterward is less than the time to update with the indexes in place. (The time cost of updating indexes is discussed under “Time Costs of Indexes” on page 10-20.)

Second, newly made indexes are the most efficient ones. Frequent updates tend to dilute the index structure, causing it to contain many partly full leaf pages. This reduces the effectiveness of an index and wastes disk space.

As another timesaving measure, make sure that a batch-updating program calls for rows in the sequence defined by the primary-key index. That sequence causes the pages of the primary-key index to be read in order and only one time each.

The presence of indexes also slows down the population of tables when you use the LOAD statement or the **dbload** utility. Loading a table that has no indexes is a very quick process (little more than a disk-to-disk sequential copy), but updating indexes adds a great deal of overhead.

The fastest way to load a table is as described in the following list:

1. Drop the table (if it exists).
2. Create the table without specifying any unique constraints.
3. Load all rows into the table.
4. Alter the table to apply the unique constraints.
5. Create the non-unique indexes.

If you cannot guarantee that the loaded data satisfies all unique constraints, you must create the unique indexes before you load the rows. It saves time if the rows are presented in the correct sequence for at least one of the indexes (if you have a choice, make it the one with the largest key). This minimizes the number of leaf pages that must be read and written.

Clustered Indexes

The term *clustered index* is a misnomer. The index is nothing special; it is the table that is modified so that its rows are physically ordered to agree with the sequence of entries in the index. (Do not confuse the clustered index with an *optical cluster*, which is a method for storing logically related blobs together on an optical volume. For information on optical clusters, see the *INFORMIX-OnLine/Optical User Manual*.)

When you know that a table is ordered by a certain index, you can take advantage of the knowledge to avoid sorting. You can also be sure that when the table is searched on that column, it is read (effectively) in sequential order instead of nonsequentially. These points are covered in Chapter 13, “Optimizing Your Queries.”

In the **stores6** database, the **orders** table has an index, **zip_ix**, on the zip code column. The following command causes the database server to put the rows of the **customer** table into descending order by zip code:

```
ALTER INDEX zip_ix TO CLUSTER
```

To cluster a table on a nonindexed column, you must create an index. The following command reorders the **orders** table by order date:

```
CREATE CLUSTERED INDEX o_date_ix ON orders (order_date ASC)
```

To reorder a table, the database server must copy the table. In the preceding example, the database server reads all rows of the table and constructs an index. Then it reads the index entries in sequence. For each entry, it reads the matching row of the table and copies it to a new table. The rows of the new table are in the desired sequence. This new table replaces the old table.

Clustering is not preserved when you alter a table. When you insert new rows, they are stored physically at the end of the table regardless of their contents. When you update rows and change the value of the clustering column, the rows are written back into their original location in the table.

When clustering is disturbed by updates, it can be restored. The following command reorders the table to restore the physical sequence:

```
ALTER INDEX o_date_ix TO CLUSTER
```

Reclustering is usually quicker than the original clustering because reading out the rows of a nearly clustered table is almost a sequential scan.

Clustering and reclustering take a lot of space and time. You can avoid some clustering by building the table in the desired order in the first place. The physical order of rows is their insertion order, so if the table is initially loaded with ordered data, no clustering is needed.

Denormalizing

The entity-relationship data model described in Chapter 8, “Building Your Data Model,” produces tables that contain no redundant or derived data; tables that are well-structured by the tenets of relational theory.

Sometimes, to meet extraordinary demands for high performance, you might have to modify the data model in ways that are undesirable from a theoretical standpoint. This section describes some modifications and their associated costs.

Shorter Rows for Faster Queries

As a general principle, tables with shorter rows yield better performance than ones with longer rows. This is because disk I/O is performed in pages, not in rows. The shorter the rows of a table, the more rows occur on a page. The more rows per page, the fewer I/O operations it takes to read the table sequentially, and the more likely it is that a nonsequential access can be satisfied from a buffer.

The entity-relationship data model put all the attributes of one entity into a single table for that entity. For some entities, this can produce rows of awkward length. You can shorten the rows. As the rows get shorter, query performance should improve.

Expelling Long Strings

The most bulky attributes are often character strings. If you can remove them from the entity table, the rows become shorter.

Using VARCHAR Strings

An existing database might contain CHAR columns that can be converted profitably to VARCHAR. VARCHAR columns shorten the average row when the average value in the CHAR column is at least 2 bytes shorter than the

existing fixed width of the column. (When NLS is enabled, NCHAR columns may replace CHAR columns, and NVARCHAR columns may replace VARCHAR columns.)

This CHAR-VARCHAR (or NCHAR-NVARCHAR) substitution does not harm the theoretical characteristics of the model. Furthermore, VARCHAR data is immediately compatible with most existing programs, forms, and reports. (Forms must be recompiled. Forms and reports should, of course, be tested on a sample database.)

Changing Long Strings to TEXT

When the typical string fills half a disk page or more, consider converting it to a TEXT column in a separate blob space. The column within the row page is only 56 bytes long, which should put many more rows on a page. However, the TEXT data type is not automatically compatible with existing programs. The code for fetching a TEXT value is more complicated than the code for fetching a CHARACTER value into a program.

Building a Symbol Table of Repeated Strings

If a column contains strings that are not unique in each row, you can remove those strings to a table in which only unique copies are stored.

For example, the **customer.city** column contains city names. Some city names are repeated down the column, and most rows have some trailing blanks in the field. Using the VARCHAR data type eliminates the blanks but not the duplication.

You can create a table named **cities**, as shown in the following example:

```
CREATE TABLE cities
(
  city_num SERIAL PRIMARY KEY,
  city_name VARCHAR(40) UNIQUE
)
```

You can change the definition of the **customer** table so that its **city** column becomes a foreign key that references the **city_num** column in the **cities** table.

You must change any program that inserts a new row into **customer** to insert the city of the new customer into **cities**. The database server return code in the SQLCODE field can indicate that the insert failed because of a duplicate key. It is not a logical error; it simply means that some existing customer is

located in that city. (However, a 4GL program must use the WHENEVER statement to trap errors; otherwise the negative value in SQLCODE terminates the program.)

Besides changing programs that insert data, you also must change all programs and stored queries that retrieve the city name. They must use a join into the new **cities** table to obtain their data. The extra complexity in programs that insert rows and the extra complexity in some queries is the cost of giving up theoretical correctness in the data model. Before you make the change, be sure it returns a reasonable savings in disk space or execution time.

Moving Strings to a Companion Table

Strings less than half a page long waste disk space if you treat them as TEXT, but you can remove them from the main table to a companion table. The use of companion tables is the subject of the next section.

Splitting Wide Tables

Consider all the attributes of an entity whose rows are too wide for good performance. Look for some theme or principle on which they can be divided into two groups. Split the table into two, a primary table and a companion table, repeating the primary key in each one. The shorter rows allow each table to be queried or updated more quickly.

Division by Bulk

One principle on which you can divide an entity table is bulk: Move the bulky attributes, which are usually character strings, to the companion table. Keep the numeric and other small attributes in the primary table. In the demonstration database, you can split the **ship_instruct** column from the **orders** table. You can call the companion table **orders_ship**. It has two columns, a primary key that is a copy of **orders.order_num** and the original **ship_instruct** column.

Division by Frequency of Use

Another principle for division of an entity is frequency of use. If a few attributes are rarely queried, they can be moved out to a companion table. In the demonstration database, it could be that the **ship_instruct**, **ship_weight**, and **ship_charge** columns are queried only in one program. In that case, you can move them out to a companion table.

Division by Frequency of Update

Updates take longer than queries, and the updating programs lock index pages and rows of data, slowing down the programs that only query. If certain attributes of an entity are updated frequently and many others are changed only rarely or not at all—and if many queries select the latter and do not need the former—you can split the volatile columns to a companion table. The updating programs contend for the use of that table, and the query programs use the primary table.

Costs of Companion Tables

Splitting a table consumes extra disk space and adds complexity. Two copies of the primary key occurs for each row, one copy in each table. Two primary-key indexes also exist. You can use the methods described in earlier sections to estimate the number of pages added.

You must modify existing programs, reports, and forms that use `SELECT *` because fewer columns are returned. Programs, reports, and forms that use attributes from both tables must perform a join to bring them together.

In this case, when you insert or delete a row, two tables are altered instead of one. If you do not coordinate the alteration of the two tables (by making them within a single transaction, for example), you lose semantic integrity.

Splitting Tall Tables

A large table poses large management problems. The 400-megabyte table takes a long time to query, of course, and it also takes a long time to back up or restore. Sorting becomes radically slower because the sort goes to three or more merge levels; sorting can even become impossible because too much disk space is required. Indexes become less effective because the index structure has three or more intermediate levels.

Consider splitting such a table into segments. Each segment has the same arrangement of columns, indexes, and constraints but a different subset of rows. The division into groups of rows should be based on the value of an attribute that appears in most queries (probably the primary key) so that a program or user can easily tell in which subtable a row belongs. For example, if the primary key is a number, rows can be allocated to 10 subtables based on a middle digit.

If rows include a time or date attribute, it might be useful to split a table into segments by age, especially if recent rows are used more often than older ones.

The benefits of splitting a large table are that the segments can be treated separately for copying, sorting, indexing, archiving, and restoring. All these operations become easier. The segments can be assigned to tablespaces in different ways to make the best use of disk devices. And, as long as most queries are based on the attribute used for segmenting, queries should be dramatically faster.

The drawback of splitting a table is that operations that apply to the whole table become much more complicated. If a query cannot be directed to one subtable, it must be written in the form of a UNION of many subqueries, each on a different subtable. This greatly complicates the design of reports and browsing programs.

Redundant and Derived Data

The data model produced by the methods of Chapter 8, “Building Your Data Model,” contains no redundant data. Every attribute appears in only one table. The data model also contains no derived data; that is, data that can be computed from existing attributes is selected as an expression based on those attributes.

These features minimize the amount of disk space used and make updating the tables as easy as possible. However, they can force you to use joins and aggregate functions often, and that may take more time than is acceptable.

As an alternative, you can introduce new columns that contain redundant or derived data, provided you understand the risks.

Adding Derived Data

In the **stores6** database, no column occurs in the **orders** table for the total price of an order. The reason is that the information can be derived from the rows of the **items** table. A program that wants to report the date and total price of order number 1009 can obtain it with the following query:

```
SELECT order_date, SUM (total_price)
FROM orders, items
WHERE orders.order_num = 1009
      AND orders.order_num = items.order_num
GROUP BY orders.order_num, orders.order_date
```

Although the join to the **items** table reads only three or four additional pages, that might be too much time for an interactive program. One solution is to add an order-total column to the **orders** table.

The costs of derived data are in disk space, complexity, and data integrity.

The disk space devoted to an order-total column is wasted because the same information is stored twice. Also, the presence of the column makes the rows of the table wider; fewer of them occur in a page and querying the table becomes slower. Most important, any program that updates the base attributes must be changed to also update the derived column. Inevitably, times occur when the derived column is wrong, when it does not contain the right derived value.

Derived data is not reliable data. In the example of a derived order price, the unreliability occurs while an order is being entered. When rows are added to the **items** table, times occur when the order total in **orders** is not equal to the sum of the corresponding rows from **items**. In general, before you allow derived data to appear in the database, you must define its accuracy as carefully as possible; actually, you should define as closely as possible the conditions under which the derived column is *unreliable*.

Adding Redundant Data

A correct data model avoids redundancy by keeping any attribute only in the table for the entity it describes. If the attribute data is needed in a different context, you make the connection by joining tables. But joining takes time. If a frequently used join affects performance, you can eliminate it by duplicating the joined data in another table.

In the demonstration database, the **manufact** table contains the names of manufacturers and their delivery times. (In a real database, it would contain many other attributes of a supplier, such as address, sale representative name, and so on.)

The contents of **manufact** are primarily a supplement to the **stock** table. Suppose that a time-critical application frequently refers to the delivery lead time of a particular product but to no other column of **manufact**. For each such reference, the database server must read two or three pages of data to perform the lookup.

You can add a new column, **lead_time**, to the **stock** table and fill it with copies of the **lead_time** column from the corresponding rows of **manufact**. That eliminates the lookup, which speeds up the application.

As with derived data, redundant data takes space and poses an integrity risk. In the example, there is not one extra copy of each manufacturer's lead-time, there are many. (Each manufacturer can appear in **stock** many times.) The programs that insert or update a row of **manufact** must be changed to update multiple rows of **stock** as well.

The integrity risk is simply that the redundant copies of the data might not be accurate. If a lead time is changed in **manufact**, the **stock** column is out-dated until it, too, is updated. As with derived data, you should take pains to define the conditions under which the redundant data might be wrong.

Maximizing Concurrency

Some databases are used by only one program at a time; others are used concurrently by multiple programs. The following factors make concurrent programs inherently slower than doing the same amount of work serially:

- Multiple programs interfere with the use of buffers and disk drives. The pages read for one program might be driven from the buffers used by a query from the next program and might have to be read again. Disk I/O for one query displaces the disk-access arm, slowing the sequential access of another program.
- Programs that modify data lock pages, delaying all other programs that use the same data.

Easing Contention

Contention is inevitable between programs using the same resources. You can deal with it in the following ways:

1. Make programs use fewer resources, either by making them do less work or by making them work more efficiently.
2. Arrange the resources better, for instance, by allocating tables to dbspaces to minimize contention.
3. Supply more resources: more memory, more and faster disk drives, and more and faster computers.

The first point is the subject of Chapter 13, “Optimizing Your Queries,” which deals with making queries faster. Sometimes you also must consider making queries do less; that is, reducing the functions available to on-line users. You can look for any allowed transaction that entails scanning an entire large table, such as any kind of summation, average, or management report, especially if it requires a join. Consider removing such transactions from on-line programs. Instead, offer a new facility that interactively schedules an off-peak job, with the output returned the next day.

Rescheduling Modifications

To the greatest extent possible, schedule modifications for times when no interactive users are using the database. One reason for this is that concurrent modifications must be done with all indexes in place; that means they incur all the time costs noted earlier. Another reason is that modifications not only lock table rows they also lock index pages. That increases the number of locks in use and the number of lock delays for all users.

The means of rescheduling modifications depend entirely on the details of your application.

Using an Update Journal

Instead of performing updates as the data becomes available, consider creating an *update journal*. This is a table that contains rows that represent pending updates. Its rows contain the following information:

- Data items to be changed in some base table, with null meaning no change
- Audit information that allows an erroneous update to be traced back to the transaction that entered the row

It is usually much faster to insert a new row in an update journal than actually to perform the update on one or more base tables. For one thing, the journal has, at most, one index to update, but the base tables usually have several. For another, locking delays are almost nil because inserting a row in a table with no indexes requires only a lock on the inserted row. Another program can insert its row even before the first program commits its transaction.

Also, because no updates are performed on the primary tables, their pages are not locked. This allows query programs to run without delays.

After peak-usage hours (perhaps after close of business), you run a batch program to validate and apply the updates, row by row as they were inserted. You obviously would want to take pains to see that no update is forgotten or applied twice, regardless of possible system failures.

One way to update the base table is to use two cursors. The first cursor is a hold cursor; it is used to scan the rows of the journal. (See “Hold Cursors” in Chapter 7.) For each row of the journal, the program goes through the following steps:

1. Issue the BEGIN WORK statement.
2. Fetch the rows from the tables being updated using an update cursor. (This locks only these rows.)
3. Validate the update information from the journal against the data in the target rows.
4. Apply the updates to the target tables.
5. Update the journal row in some way to mark it as finished.
6. Issue the COMMIT WORK statement (or the ROLLBACK WORK statement, if an error is found).

You run a different program to drop and re-create the journal table only after every journal row is validated, applied, and marked.

The obvious disadvantage of an update journal is that the base tables do not reflect the most current data. If it is essential that updates be instantly visible, the journal scheme does not work.

The great advantages of reduced I/O and reduced locking delays during peak hours are a powerful argument in favor of an update journal. Deferred updates are accepted in many applications. For example, no bank promises to know your account balance more precisely than as of the close of business the preceding day, and for exactly this reason: the bank records transactions in a journal during the day and applies the updates overnight.

Isolating and Dispersing Updates

If updates really must be performed interactively during peak hours, you must find a different way to isolate queries from updates.

Splitting Tables to Isolate Volatile Columns

This idea was covered in an earlier section. If it is necessary to perform updates during peak hours, examine the structure of the tables. Can you separate the columns into volatile ones that are updated constantly and static ones that are rarely updated? If so, consider splitting the tables so that the volatile columns are isolated in a companion table.

Depending on the mix of operations and on the priorities of the users, you can then put either the static or the volatile subtable on your fastest disk drive.

Dispersing Bottleneck Tables

Small tables are sometimes used for summaries, usage records, and audits. For example, your interactive programs can maintain a table with a row for each authorized user. Each time the user starts or ends a program, it updates that user's row to show time on, time off, number of transactions, or other work-monitoring data.

Small tables that are only read do not cause performance problems, but small tables that are the subject of concurrent updates do. Any small table that must be updated on every transaction can become a bottleneck in which every on-line program queues up and waits.

To eliminate the problem, either use a journal with off-peak updates as described earlier, or disperse the bottleneck by creating many tables. To monitor users, create a one-row table for each user.

Summary

When tables are moderate in size and only one user at a time accesses the database, carefully applied relational theory is sufficient to produce good performance.

When both the number of tables and the number of users become larger and the response time begins to degrade, you must turn to practical solutions.

The first step is to understand and take advantage of the tools that **INFORMIX-OnLine Dynamic Server** offers. It permits you to arrange your tables on the hardware for maximum benefit. Then, one step at a time and always measuring, you can begin complicating the structure of the data model and the programs that use it.

Granting and Limiting Access to Your Database

Chapter Overview	3
Controlling Access to Databases	4
Securing Database Files	4
Multiuser Systems	4
Single-User Systems	4
Securing Confidential Data	5
Granting Privileges	5
Database-Level Privileges	5
Connect Privilege	6
The Users and the Public	6
Resource Privilege	7
Database Administrator Privilege	7
Ownership Rights	7
Table-Level Privileges	7
Access Privileges	8
Index, Alter, and References Privileges	8
Privileges in the System Catalog	9
Column-Level Privileges	10
Procedure-Level Privileges	12
Automating Privileges	13
Automating with INFORMIX-4GL	13
Automating with a Command Script	14
Controlling Access to Data Using Stored Procedures	15
Restricting Reads of Data	16
Restricting Changes to Data	16

Monitoring Changes to Data	17
Restricting Object Creation	18
Using Views	19
Creating Views	20
Duplicate Rows from Views	22
Restrictions on Views	22
When the Basis Changes	22
Modifying Through a View	23
Deleting Through a View	24
Updating a View	24
Inserting into a View	25
Using WITH CHECK OPTION	25
Privileges and Views	27
Privileges When Creating a View	27
Privileges When Using a View	28
Summary	30

Chapter Overview

In some databases, all data is accessible to every user. In others, this is not the case; some users are denied access to some or all of the data. You can restrict access to data at the following levels, which are the subject of this chapter:

- When the database is stored in operating system files, you can sometimes use the file-permission features of the operating system.

This level is not available when **INFORMIX-OnLine Dynamic Server** holds the database. It manages its own disk space and the rules of the operating system do not apply.

- You can use the GRANT and REVOKE statements to give or deny access to the database or to specific tables, and you can control the kinds of uses that people can make of the database.
- You can use the CREATE PROCEDURE statement to write and compile a stored procedure that controls and monitors which users can read, modify, or create database tables.
- You can use the CREATE VIEW statement to prepare a restricted or modified view of the data. The restriction can be vertical, excluding certain columns, or horizontal, excluding certain rows, or both.
- You can combine GRANT and CREATE VIEW statements to achieve precise control over the parts of a table a user can modify and with what data.

In addition to these points, **INFORMIX-OnLine/Secure** offers a type of automatic security called mandatory access control (MAC). With MAC, all users and all data are assigned a security label, and **OnLine/Secure** compares the labels before allowing access. For example, a TOP SECRET user could view an UNCLASSIFIED row, but an UNCLASSIFIED user could not view a TOP SECRET row. This topic, and other topics related to security in **OnLine/Secure** are addressed in the *INFORMIX-OnLine/Secure Security Features User's Guide*.

Controlling Access to Databases

The normal database-privilege mechanisms are based on the GRANT and REVOKE statements. They are covered in the section “Granting Privileges” on page 11-5. However, you can sometimes use the facilities of the operating system as an additional way to control access to a database.

Securing Database Files

Database servers other than **INFORMIX-OnLine Dynamic Server** store databases in operating system files. Typically, a database is represented as a number of files: one for each table, one for the indexes on each table, and possibly others. The files are collected in a directory. The directory represents the database as a whole.

Multiuser Systems

You can deny access to the database by denying access to the database directory. The means by which you can do this depend on your operating system and your computer hardware. Multiuser operating systems have software facilities such as UNIX file permissions.

***Note:** In UNIX, the database directory is created with group identity **informix**, and the database server always runs under group identity **informix**. Thus, you cannot use group permissions to restrict access to a particular group of users. You can, however, only remove all group permissions (file mode 700) and deny access to anyone except the owner of the directory.*

You also can deny access to individual tables in this way; for example, by making the files that represent those tables unavailable to certain users, while leaving the rest of the files accessible. However, the database servers are not designed for tricks of this kind. When an unauthorized user tries to query one of the tables, the database server probably returns an error message about not being able to locate a file. This may confuse users.

Single-User Systems

Typical single-user systems have few software controls on file access; you can make a database inaccessible to others only by writing it on a disk that you can detach from the machine and keep locked.

None of these techniques apply when you use the **INFORMIX-OnLine Dynamic Server** database server. It controls its own disk space at the device level, bypassing the file-access mechanisms of the operating system.

Securing Confidential Data

No matter what access controls the operating system gives you, when the contents of an entire database are highly sensitive you might not want to leave it on a public disk fixed to the machine. You can circumvent normal software controls when the data must be secure.

When the database is not being used by you or another authorized person, it does not have to be available on-line. You can make it inaccessible in one of the following ways, which have varying degrees of inconvenience:

- Detach the physical medium from the machine, and take it away. If the disk itself is not removable, the disk drive might be removable.
- Copy the database directory to tape, and take possession of the tape.
- Copy the database files using an encryption utility. Keep only the encrypted version.

In the latter two cases, after making the copies, you must remember to erase the original database files using a program that overwrites an erased file with null data.

Instead of removing the entire database directory, you can copy and then erase the files that represent individual tables. However, do not overlook the fact that index files contain copies of the data from the indexed column or columns. Remove and erase the index files as well as the table files.

Granting Privileges

The authorization to use a database is called a *privilege*. For example, the authorization to use a database at all is called the Connect privilege, and the authorization to insert a row into a table is called the Insert privilege. You control the use of a database by granting these privileges to other users or by revoking them.

Two groups of privileges exist that control the actions a user can perform on data. These are database-level privileges, which affect the entire database, and table-level privileges, which relate to individual tables. In addition to these two groups, procedure-level privileges exist, which determine who can execute a procedure.

Database-Level Privileges

The three levels of database privilege provide an overall means of controlling who accesses a database.

Connect Privilege

The least of the privilege levels is Connect, which gives a user the basic ability to query and modify tables. Users with the Connect privilege can perform the following functions:

- Execute the SELECT, INSERT, UPDATE, and DELETE statements, provided that they have the necessary table-level privileges.
- Execute a stored procedure, provided that they have the necessary table-level privileges.
- Create views, provided that they are permitted to query the tables on which the views are based.
- Create temporary tables and create indexes on the temporary tables.

The Users and the Public

Privileges are granted to single users by name or to all users under the name of PUBLIC. Any grant to the public serves as a default privilege.

Prior to executing a statement, the database server determines whether the user has the necessary privileges. (The information is in the system catalog; see “Privileges in the System Catalog” on page 11-9.)

The database server looks first for privileges granted specifically to the requesting user. If it finds such a grant, it uses that information. It then checks to see if less restrictive privileges have been granted to PUBLIC. If so, the database server uses the less-restrictive privileges. If there has been no grant to that user, the database server looks for privileges granted to public. If it finds a relevant one, it uses that.

Thus, you can set a minimum level of privilege for all users by granting privileges to public. You can override that in specific cases by granting higher individual privileges to users.

The Connect privilege is necessary before users can access a database. Ordinarily, in a database that does not contain highly sensitive or private data, you give the GRANT CONNECT TO PUBLIC privilege shortly after creating the database.

If you do not grant the Connect privilege to public, the only users who can access the database through the database server are those to whom you specifically grant the Connect privilege. If limited users should have access, this is your means of providing it to them and denying it to all others.

Resource Privilege

The Resource privilege carries the same authorization as the Connect privilege. In addition, users with the Resource privilege can create new, permanent tables, indexes, and stored procedures, thus permanently allocating disk space.

Database Administrator Privilege

The highest level of database privilege is *Database Administrator*, or DBA. When you create a database, you are automatically the DBA. Holders of the DBA privilege can perform the following functions:

- Execute the DROP DATABASE, START DATABASE, and ROLLFORWARD DATABASE statements.
- Drop or alter any object regardless of who owns it.
- Create tables, views, and indexes to be owned by other users.
- Grant database privileges, including the DBA privilege, to another user.
- Alter the NEXT SIZE (but no other attribute) of the system catalog tables, and insert, delete, or update rows of any system catalog table except **systables**.



Warning: Although users with the DBA privilege can modify the system catalog tables, Informix strongly recommends that you do not update, delete, or alter any rows in the system catalog tables. Modifying the system catalog tables can destroy the integrity of the database.

Ownership Rights

The database, and every table, view, index, procedure, and synonym in it, has an owner. The owner of an object is usually the person who created it, although a user with the DBA privilege can create objects to be owned by others.

The owner of an object has all rights to that object and can alter or drop it without needing additional privileges.

Table-Level Privileges

You can apply seven privileges, table by table, to allow nonowners the privileges of owners. Four of them—the Select, Insert, Delete, and Update privileges—control access to the contents of the table. The Index privilege

controls index creation. The Alter privilege controls the authorization to change the table definition. The References privilege controls the authorization to specify referential constraints on a table.

In an ANSI-compliant database, only the table owner has any privileges. In other databases, the database server, as part of creating a table, automatically grants all table privileges—except Alter and References—to public. This means that a newly created table is accessible to any user with the Connect privilege. If this is not what you want—if users exist with the Connect privilege who should not be able to access this table—you must revoke all privileges on the table from public after you create the table.

Access Privileges

Four privileges govern how users can access a table. As the owner of the table, you can grant or withhold the following privileges independently:

- The Select privilege allows selection, including selecting into temporary tables.
- The Insert privilege allows a user to add new rows.
- The Update privilege allows a user to modify existing rows.
- The Delete privilege allows a user to delete rows.

The Select privilege is necessary for a user to retrieve the contents of a table. However, The Select privilege is not a precondition for the other privileges. A user can have Insert or Update privileges without having the Select privilege.

For example, your application might have a usage table. Every time a certain program is started, it inserts a row into the usage table to document that it was used. Before the program terminates, it updates that row to show how long it ran and perhaps to record counts of work performed by its user.

If you want any user of the program to be able to insert and update rows in this usage table, you grant Insert and Update privileges on it to public. However, you might grant the Select privilege to only a few users.

Index, Alter, and References Privileges

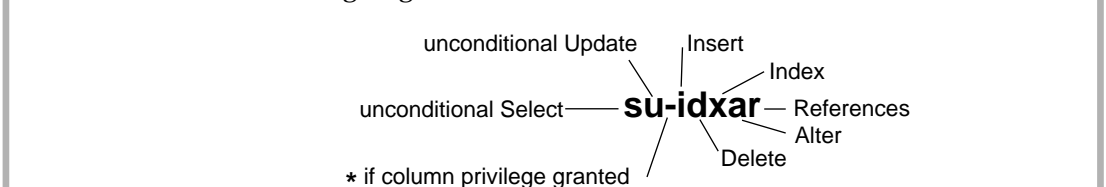
The Index privilege permits its holder to create and alter indexes on the table. The Index privilege, similar to the Select, Insert, Update, and Delete privileges, is granted automatically to public when a table is created.

Privileges in the System Catalog

Privileges are recorded in the system catalog tables. Any user with the Connect privilege can query the system catalog tables to determine what privileges have been granted and to whom.

Database privileges are recorded in the **sysusers** table, in which the primary key is user-id and the only other column contains a single character C, R, or D for the privilege level. A grant to the keyword of PUBLIC is reflected as a user name of **public** (lowercase).

Table-level privileges are recorded in **systabauth**, which uses a composite primary key of the table number, grantor, and grantee. In the **tabauth** column, the privileges are encoded in a six-letter list as shown in the following diagram:



A hyphen means an ungranted privilege, so that a grant of all privileges is shown as `su-idxar`, and `-u-----` shows a grant of only Update. The code letters are normally lowercase, but they are uppercase when the keywords `WITH GRANT OPTION` are used in the `GRANT` statement.

When an asterisk appears in the third position, some column-level privilege exists for that table and grantee. The specific privilege is recorded in **syscolauth**. Its primary key is a composite of the table number, the grantor, the grantee, and the column number. The only attribute is a three-letter list showing the type of privilege: **s**, **u**, or **r**.

You can grant the Index privilege to anyone, but to exercise the ability, the user must also hold the Resource database privilege. So, although the Index privilege is granted automatically (except in ANSI-compliant databases), users who have only the Connect privilege to the database cannot exercise their Index privilege. This is reasonable because an index can fill a large amount of disk space.

The Alter privilege permits its holder to use the ALTER TABLE statement on the table, including the power to add and drop columns, reset the starting point for SERIAL columns, and so on. You should grant the Alter privilege only to users who understand the data model very well and whom you trust to exercise their power very carefully.

The References privilege allows you to impose referential constraints on a table. As with the Alter privilege, you should grant the References privilege only to users who understand the data model very well.

Column-Level Privileges

You can qualify the Select, Update, and References privileges with the names of specific columns. This allows you to grant very specific access to a table: You can permit a user to see only certain columns, you can allow a user to update only certain columns, or you can allow a user to impose referential constraints on certain columns.

Using **INFORMIX-OnLine Dynamic Server** (so that table data can be inspected only through a call to the database server), this feature solves the problem posed earlier: that only certain users should know the salary, performance review or other sensitive attributes of an employee. To make the example specific, suppose a table of employee data is defined as shown in Figure 11-1.

```
CREATE TABLE hr_data
(
  emp_key INTEGER,
  emp_name CHAR(40),
  hire_date DATE,
  dept_num SMALLINT,
  user-id CHAR(18),
  salary DECIMAL(8,2)
  performance_level CHAR(1),
  performance_notes TEXT
)
```

Figure 11-1 *A table of confidential employee information*

Because this table contains sensitive data, you execute the following statement immediately after creating it:

```
REVOKE ALL ON hr_data FROM PUBLIC
```

For selected persons in the Human Resources department and for all managers, you execute a statement such as the following one:

```
GRANT SELECT ON hr_data TO harold_r
```

In this way, you permit certain users to view all columns. (The final section of this chapter discusses a way to limit the view of managers to only their own employees.) For the first-line managers who carry out performance reviews, you could execute a statement such as the following one:

```
GRANT UPDATE (performance_level, performance_notes)
ON hr_data TO wallace_s, margot_t
```

This statement permits the managers to enter their evaluations of their employees. Only for the manager of the Human Resources department or whomever is trusted to alter salary levels, would you execute a statement such as the following one:

```
GRANT UPDATE (salary) ON hr_data TO willard_b
```

For the clerks in the Human Resources department, you could execute a statement such as the following one:

```
GRANT UPDATE (emp_key, emp_name, hire_date, dept_num)
ON hr_data TO marvin_t
```

This statement gives certain users the ability to maintain the nonsensitive columns but denies them authorization to change performance ratings or salaries. The person in the MIS department who assigns computer user-ids is the beneficiary of a statement such as the following one:

```
GRANT UPDATE (user_id) ON hr_data TO eudora_b
```

On behalf of all users who are allowed to connect to the database but who are not authorized to see salaries or performance reviews, you execute statements such as the following one to permit them to see the nonsensitive data:

```
GRANT SELECT (emp_key, emp_name, hire_date, dept_num, user-id)
ON hr_data TO george_b, john_s
```

These users can perform queries such as the following one:

```
SELECT COUNT(*) FROM hr_data WHERE dept_num IN (32,33,34)
```

However, any attempt to execute a query such as the following one produces an error message and no data:

```
SELECT performance_level FROM hr_data
WHERE emp_name LIKE '*Smythe'
```

Procedure-Level Privileges

You can apply the Execute privilege on a procedure to authorize nonowners to run a procedure. If you create a procedure in a database that is not ANSI-compliant, the default procedure-level privilege is Public; you do not need to grant the Execute privilege to specific users unless you have first revoked it. If you create a procedure in an ANSI-compliant database, no other users have the Execute privilege by default; you must grant specific users the Execute privilege. The following example grants the Execute privilege to the user **orion** so that **orion** can use the stored procedure named **read-address**:

```
GRANT EXECUTE ON read_address TO orion;
```

Procedure-level privileges are recorded in the **sysprocauth** system catalog table. The **sysprocauth** table uses a primary key of the procedure number, grantor, and grantee. In the **procauth** column, the execute privilege is indicated by a lowercase letter “e.” If the execute privilege was granted with the WITH GRANT option, the privilege is represented by an uppercase letter “E.”

For more information on procedure-level privileges see “Privileges on Stored Procedures” on page 14-13.

Automating Privileges

It might seem that this design forces you to execute a tedious number of GRANT statements when you first set up the database. Furthermore, privileges require constant maintenance as people change their jobs. For example, if a clerk in Human Resources is terminated, you want to revoke the Update privilege as soon as possible; otherwise you risk the unhappy employee executing a statement such as the following one:

```
UPDATE hr_data
  SET (emp_name, hire_date, dept_num) = (NULL, NULL, 0)
```

Less dramatic, but equally necessary, changes of privilege are required daily, or even hourly, in any model that contains sensitive data. If you anticipate this need, you can prepare some automated tools to help maintain privileges.

Your first step should be to specify privilege classes that are based on the jobs of the users, not on the structure of the tables. For example, a first-line manager needs the following privileges:

- The Select and limited Update privilege on the hypothetical **hr_data** table
- The Connect privilege to this and other databases
- Some degree of privilege on several tables in those databases

When the manager is promoted to a staff position or sent to a field office, you must revoke all those privileges and grant a new set of privileges.

Define the privilege classes you support, and for each class specify the databases, tables, and columns to which you must give access. Then devise two automated procedures for each class: one to grant the class to a user and one to revoke it.

Automating with INFORMIX-4GL

The mechanism you use depends on your operating system and other tools. If you are a programmer, the most flexible tool is probably **INFORMIX-4GL**. **4GL** makes it quite easy to program a simple user interaction, as shown in the following fragment:

```
DEFINE mgr_id char(20)
PROMPT 'What is the user-id of the new manager? ' FOR mgr_id
CALL table_grant ('SELECT', 'hr_data', mgr_id)
```

Unfortunately, although **INFORMIX-4GL** allows you to mix GRANT and REVOKE statements freely with other program statements, it does not permit you to create parameters from them from program variables. To customize a GRANT statement with a user-id taken from user input, the program must build the statement as a string, prepare it with a PREPARE statement, and execute it with an EXECUTE statement. (These statements are discussed in detail in Chapter 5, "Programming with SQL," where the following example is analyzed in detail.)

Figure 11-2 shows one possible definition of the 4GL function **table_grant()** that is invoked by the CALL statement in the preceding example.

```
FUNCTION table_grant (priv_to_grant, table_name, user_id)
  DEFINE  priv_to_grant char(100), {may include column-list}
         table_name CHAR(20),
         user_id CHAR(20),
         grant_stmt CHAR(200)
  LET grant_stmt = ' GRANT ', priv_to_grant,
                  ' ON ', table_name,
                  ' TO ', user_id
  WHENEVER ERROR CONTINUE
  PREPARE the_grant FROM grant_stmt
  IF status = 0 THEN
    EXECUTE the_grant
  END IF
  IF status <> 0 THEN
    DISPLAY 'Sorry, got error #', status, 'attempting:'
    DISPLAY '      ', grant_stmt
  END IF
  WHENEVER ERROR STOP
END FUNCTION
```

Figure 11-2 *A 4GL function that builds, prepares, and executes a GRANT statement*

Automating with a Command Script

Your operating system probably supports automatic execution of command scripts. In most operating environments, **DB-Access** and **INFORMIX-SQL**, the interactive SQL tools, accept commands and SQL statements to execute from the command line. You can combine these two features to automate privilege maintenance.

The details depend on your operating system and the version of **DB-Access** or **INFORMIX-SQL** that you are using. In essence, you want to create a command script that performs the following functions:

- Takes a user-id whose privileges are to be changed as its parameter
- Prepares a file of GRANT or REVOKE statements customized to contain that user-id
- Invokes **DB-Access** or **INFORMIX-SQL** with parameters that tell it to select the database and execute the prepared file of GRANT or REVOKE statements

In this way, you can reduce the change of the privilege class of a user to one or two commands.

Controlling Access to Data Using Stored Procedures

You can use a stored procedure to control access to individual tables and columns in the database. You can accomplish various degrees of access control through a procedure. (Stored procedures are fully described in Chapter 14, “Creating and Using Stored Procedures.”) A powerful feature of SPL is the ability to designate a stored procedure as a DBA-privileged procedure. Writing a DBA-privileged procedure, you can allow users who have few or no table privileges to have DBA privileges when they execute the procedure. In the procedure, users can carry out very specific tasks with their temporary DBA privilege. The DBA-privileged feature allows you to accomplish the following tasks:

- You can restrict how much information individual users can read from a table.
- You can restrict all the changes made to the database and ensure that entire tables are not emptied or changed accidentally.
- You can monitor an entire class of changes made to a table, such as deletions or insertions.
- You can restrict all object creation (data definition) to occur within a stored procedure so that you have complete control over how tables, indexes, and views are built.

Restricting Reads of Data

The procedure in Figure 11-3 hides the SQL syntax from users, but it requires that users have the Select privilege on the **customer** table. If you want to restrict what users can select, you can write your procedure to work in the following environment:

- You are the DBA of the database.
- The users have the Connect privilege to the database. They do not have the Select privilege on the table.
- Your stored procedure (or set of stored procedures) is created using the DBA keyword.
- Your stored procedure (or set of stored procedures) reads from the table for users.

If you want users to read only the name, address, and telephone number of a customer, you can modify the procedure as shown in Figure 11-3.

```
CREATE DBA PROCEDURE read_customer(cnum INT)
RETURNING CHAR(15), CHAR(15), CHAR(18);

DEFINE p_lname,p_fname CHAR(15);
DEFINE p_phone CHAR(18);

SELECT fname, lname, phone
      INTO p_fname, p_lname, p_phone
      FROM customer
      WHERE customer_num = cnum;

RETURN p_fname, p_lname, p_phone;

END PROCEDURE;
```

Figure 11-3 *Stored procedure to restrict reads on customer data*

Restricting Changes to Data

Using stored procedures, you can restrict changes made to a table. Simply channel all changes through a stored procedure. The stored procedure makes the changes, rather than users making the changes directly. If you want to

limit users to deleting one row at a time to ensure that they do not accidentally remove all the rows in the table, set up the database with the following privileges:

- You are the DBA of the database.
- All the users have the Connect privilege to the database. They may or may not have the Resource privilege. They do not have the Delete (for this example) privilege on the table being protected.
- Your stored procedure is created using the DBA keyword.
- Your stored procedure performs the deletion.

Write a stored procedure similar to the one shown in Figure 11-4, which deletes rows from the **customer** table using a WHERE clause with the **customer_num** provided by the user.

```
CREATE DBA PROCEDURE delete_customer(cnum INT)
DELETE FROM customer
      WHERE customer_num = cnum;
END PROCEDURE;
```

Figure 11-4 *Stored procedure to delete a row*

Monitoring Changes to Data

Using stored procedures, you can create a record of changes made to a database. You can record changes made by a particular user, or you can make a record of each time a change is made.

You can monitor all the changes made to the database by a single user. Simply channel all changes through stored procedures that keep track of changes made by each user. If you want to record each time the user **acctclrk** modifies the database, set up the database with the following privileges:

- You are the DBA of the database.
- All other users have the Connect privilege to the database. They may or may not have the Resource privilege. They do not have the Delete (for this example) privilege on the table being protected.
- Your stored procedure is created using the DBA keyword.
- Your stored procedure performs the deletion and records that a change has been made by a certain user.

Write a stored procedure similar to the one in Figure 11-6, which updates a table using a customer number provided by the user. If the user happens to be **acctclrk**, a record of the deletion is put in the file **updates**.

```
CREATE DBA PROCEDURE delete_customer(cnum INT)

DEFINE username CHAR(8);

DELETE FROM customer
      WHERE customer_num = cnum;

IF username = 'acctclrk' THEN
      SYSTEM 'echo Delete from customer by acctclrk >> /mis/records/updates' ;
END IF
END PROCEDURE;
```

Figure 11-5 *A procedure to delete rows and record changes made by a certain user*

You can monitor all the deletions made through the procedure by removing the IF statement and making the SYSTEM statement more general. If you change the procedure in Figure 11-6 to record all deletions, it looks like the procedure in Figure 11-6.

```
CREATE DBA PROCEDURE delete_customer(cnum INT)

DEFINE username CHAR(8);
LET username = USER ;
DELETE FROM tname WHERE customer_num = cnum;

SYSTEM
      'echo Deletion made from customer table, by ' || username || '>>/hr/records/deletes';

END PROCEDURE;
```

Figure 11-6 *Stored procedure to delete rows and record user*

Restricting Object Creation

To put restraints on what and how objects are built, use stored procedures within the following setting:

- You are the DBA of the database.
- All the other users have the Connect privilege to the database. They do not have the Resource privilege.
- Your stored procedure (or set of stored procedures) is created using the DBA keyword.

- Your stored procedure (or set of stored procedures) creates tables, indexes, and views in the way you defined them. You might use such a procedure to set up a training database environment.

Your procedure might include the creation of one or more tables and associated indexes, as shown in Figure 11-7.

```
CREATE DBA PROCEDURE all_objects()

CREATE TABLE learn1 (intone SERIAL, inttwo INT NOT NULL, charcol CHAR(10) )
CREATE INDEX learn_ix ON learn1 (inttwo).
CREATE TABLE toys (name CHAR(15) NOT NULL UNIQUE,
                     description CHAR(30), on_hand INT);
END PROCEDURE;
```

Figure 11-7 *DBA-mode procedure that adds tables and indexes to the database*

To use the **all_objects** procedure to control additions of columns to tables, revoke the Resource privilege on the database from all users. When users try to create a table, index, or view using an SQL statement outside your procedure, they cannot do so. When users execute the procedure, they have a temporary DBA privilege so the CREATE TABLE statement, for example, succeeds, and you are guaranteed that every column added has a constraint placed on it. In addition, objects created by users are owned by that user. For the **all_objects** procedure, the two tables and the index are owned by whoever executed the procedure.

Using Views

A *view* is a synthetic table. You can query it as if it were a table, and in some cases you can update it as if it were a table. However, it is not a table, rather, it is a synthesis of the data that exists in real tables and other views.

The basis of a view is a SELECT statement. When you create a view, you define a SELECT statement that generates the contents of the view at the time the view is accessed. A user also queries a view using a SELECT statement. The database server merges the SELECT statement of the user with the one defined for the view and then actually performs the combined statements.

The result has the appearance of a table; it is enough like a table that a view even can be based on other views, or on joins of tables and other views.

Because you write a SELECT statement that determines the contents of the view, you can use views for any of the following purposes:

- To restrict users to particular columns of tables
You name only permitted columns in the select list in the view.
- To restrict users to particular rows of tables
You specify a WHERE clause that returns only permitted rows.
- To constrain inserted and updated values to certain ranges
You can use the WITH CHECK OPTION (discussed on page 11-25) to enforce constraints.
- To provide access to derived data without having to store redundant data in the database
You write the expressions that derive the data into the select list in the view. Each time you query the view, the data is derived anew. The derived data is always up to date, yet no redundancies are introduced into the data model.
- To hide the details of a complicated SELECT statement
You hide complexities of a multitable join in the view so that neither users nor application programmers need to repeat them.

Creating Views

The following example creates a view based on a table in the **stores6** database:

```
CREATE VIEW name_only AS
SELECT customer_num, fname, lname FROM customer
```

The view exposes only three columns of the table. Because it contains no WHERE clause, the view does not restrict the rows that can appear.

The following example creates a view based on a table that is available when NLS has been enabled. In it, the view, column, and table names contain foreign characters.

```
CREATE VIEW çà_va AS
SELECT numéro, nom FROM abonnés;
```

The following example is based on the join of two tables:

```
CREATE VIEW full_addr AS
SELECT address1, address2, city, state.sname, zipcode
FROM customer, state
WHERE customer.state = state.code
```

The table of state names reduces the redundancy of the database; it allows for storing the full state names only once, which can be very useful for lengthy state names such as Minnesota. This **full_addr** view lets users retrieve the address as if the full state name were stored in every row. The following two queries are equivalent:

```
SELECT * FROM full_addr WHERE customer_num = 105

SELECT address1, address2, city, state.sname, zipcode
FROM customer, state
WHERE customer.state = state.code
AND customer_num = 105
```

However, be careful when you define views that are based on joins. Such views are not *modifiable*; that is, you cannot use them with UPDATE, DELETE, or INSERT statements. (Modifying through views is covered beginning on page 11-23.)

The following example restricts the rows that can be seen in the view:

```
CREATE VIEW no_cal_cust AS
SELECT * FROM customer WHERE NOT state = 'CA'
```

This view exposes all columns of the **customer** table, but only certain rows. The following example is a view that restricts users to rows that are relevant to them:

```
CREATE VIEW my_calls AS
SELECT * FROM cust_calls WHERE user_id = USER
```

All the columns of the **cust_calls** table are available but only in those rows that contain the user-ids of the users who execute the query.

Duplicate Rows from Views

It is possible for a view to produce duplicate rows, even when the underlying table has only unique rows. If the view SELECT statement can return duplicate rows, the view itself can appear to contain duplicate rows.

You can prevent this problem in two ways. One way is to specify DISTINCT in the select list in the view. However, that makes it impossible to modify through the view. The alternative is to always select a column or group of columns that is constrained to be unique. (You can be sure only unique rows are returned if you select the columns of a primary key or of a candidate key. Primary and candidate keys are discussed in Chapter 8, “Building Your Data Model.”)

Restrictions on Views

Because a view is not really a table, it cannot be indexed, and it cannot be the object of such statements as ALTER TABLE and RENAME TABLE. The columns of a view cannot be renamed with RENAME COLUMN. To change anything about the definition of a view, you must drop the view and re-create it.

Because it must be merged with the user’s query, the SELECT statement on which a view is based cannot contain any of the following clauses:

INTO TEMP	The user’s query might contain INTO TEMP; if the view also contains it, the data would not know where to go.
UNION	The user’s query might contain UNION. No meaning has been defined for nested UNION clauses.
ORDER BY	The user’s query might contain ORDER BY. If the view also contains it, the choice of columns or sort directions could be in conflict.

When the Basis Changes

The tables and views on which a view is based can change in several ways. The view automatically reflects most of the changes.

When a table or view is dropped, any views in the same database that depend on it are automatically dropped.

The only way to alter the definition of a view is to drop and re-create it. Therefore, if you change the definition of a view on which other views depend, you must also re-create the other views (because they all have been dropped).

When a table is renamed, any views in the same database that depend on it are modified to use the new name. When a column is renamed, views in the same database that depend on that table are updated to select the proper column. However, the names of columns in the views themselves are not changed. For an example of this, recall the following view on the **customer** table:

```
CREATE VIEW name_only AS
  SELECT customer_num, fname, lname FROM customer
```

Now suppose that the **customer** table is changed in the following way:

```
RENAME COLUMN customer.lname TO surname
```

To select last names of customers directly, you must now select the new column name. However, the name of the column as seen through the view is unchanged. The following two queries are equivalent:

```
SELECT fname, surname FROM customer

SELECT fname, lname FROM name_only
```

When you alter a table by dropping a column, views are not modified. If they are used, error -217 (Column not found in any table in the query) occurs. The reason views are not dropped is that you can change the order of columns in a table by dropping a column and then adding a new column of the same name. If you do this, views based on that table continue to work. They retain their original sequence of columns.

INFORMIX-OnLine Dynamic Server permits you to base a view on tables and views in external databases. Changes to tables and views in other databases are not reflected in views. Such changes might not be apparent until someone queries the view and gets an error because an external table changed.

Modifying Through a View

It is possible to modify views as if they were tables. Some views can be modified and others not, depending on their SELECT statements. The restrictions are different, depending on whether you use DELETE, UPDATE, or INSERT statements.

No modification is possible on a view when its SELECT statement contains any of the following features:

- A join of two or more tables
Many anomalies arise if the database server tries to distribute modified data correctly across the joined tables.
- An aggregate function or the GROUP BY clause
The rows of the view represent many combined rows of data; the database server cannot distribute modified data into them.
- The DISTINCT keyword or its synonym UNIQUE
The rows of the view represent a selection from among possibly many duplicate rows; the database server cannot tell which of the original rows should receive the modification.

When a view avoids all these things, each row of the view corresponds to exactly one row of one table. Such a view is *modifiable*. (Of course, particular users can modify a view only if they have suitable privileges. Privileges on views are discussed beginning on page 11-27.)

Deleting Through a View

A modifiable view can be used with a DELETE statement as if it were a table. The database server deletes the proper row of the underlying table.

Updating a View

You can use a modifiable view with an UPDATE statement as if it were a table. However, a modifiable view can still contain derived columns; that is, columns that are produced by expressions in the select list of the CREATE VIEW statement. You cannot update derived columns (sometimes called *virtual* columns).

When a column is derived from a simple arithmetic combination of a column with a constant value (for example, `order_date + 30`), the database server can, in principle, figure out how to invert the expression (in this case, by subtracting 30 from the update value) and perform the update. However, much more complicated expressions are possible, most of which cannot easily be inverted. Therefore, the database server does not support updating any derived column.

Figure 11-8 shows a modifiable view that contains a derived column, and an UPDATE statement that can be accepted against it.

```
CREATE VIEW call_response(user_id,received,resolved,duration)AS
  SELECT user_id,call_dtime,res_dtime,res_dtime-call_dtime
  FROM cust_calls
  WHERE user_id = USER

UPDATE call_response SET resolved = TODAY
  WHERE resolved IS NULL
```

Figure 11-8 *A modifiable view and an UPDATE statement*

The duration column of the view cannot be updated because it represents an expression (the database server cannot, even in principle, decide how to distribute an update value between the two columns named in the expression). But as long as no derived columns are named in the SET clause, the update can be performed as if the view were a table.

A view can return duplicate rows even though the rows of the underlying table are unique. You cannot distinguish one duplicate row from another. If you update one of a set of duplicate rows (for example, by using a cursor to update WHERE CURRENT), you cannot be sure which row in the underlying table receives the update.

Inserting into a View

You can insert rows into a view provided that the view is modifiable *and* contains no derived columns. The reason for the second restriction is that an inserted row must provide values for all columns, and the database server cannot tell how to distribute an inserted value through an expression. An attempt to insert into the **call_response** view, shown in Figure 11-8, would fail.

When a modifiable view contains no derived columns, you can insert into it as if it were a table. However, the database server uses null as the value for any column that is not exposed by the view. If such a column does not allow nulls, an error occurs and the insert fails.

Using WITH CHECK OPTION

It is possible to insert into a view a row that does not satisfy the conditions of the view; that is, a row that is not visible through the view. It is also possible to update a row of a view so that it no longer satisfies the conditions of the view.

If this is improper, you can add the clause `WITH CHECK OPTION` when you create the view. This clause asks the database server to test every inserted or updated row to ensure that it meets the conditions set by the `WHERE` clause of the view. The database server rejects the operation with an error if the conditions are not met.

In Figure 11-8, the view named **call_response** is defined as shown in the following example:

```
CREATE VIEW call_response (user_id, received, resolved, duration) AS
  SELECT user_id, call_dtime, res_dtime, res_dtime - call_dtime
  FROM cust_calls
  WHERE user_id = USER
```

It is possible to update the **user_id** column of the view, as shown in the following example:

```
UPDATE call_response SET user_id = 'lenora'
  WHERE received BETWEEN TODAY AND TODAY-7
```

The view requires rows in which **user_id** equals `USER`. If this update is performed by a user named **tony**, the updated rows vanish from the view. However, you can create the view as shown in the following example:

```
CREATE VIEW call_response (user_id, received, resolved, duration) AS
  SELECT user_id, call_dtime, res_dtime, res_dtime - call_dtime
  FROM cust_calls
  WHERE user_id = USER
WITH CHECK OPTION
```

The preceding update by **tony** is rejected as an error.

You can use the WITH CHECK OPTION feature to enforce any kind of data constraint that can be stated as a Boolean expression. In the following example, you can create a view of a table in which all the logical constraints on data are expressed as conditions of the WHERE clause. Then you can require all modifications to the table to be made through the view.

```
CREATE VIEW order_insert AS
  SELECT * FROM orders O
    WHERE order_date = TODAY -- no back-dated entries
      AND EXISTS -- ensure valid foreign key
        (SELECT * FROM customer C
          WHERE O.customer_num = C.customer_num)
      AND ship_weight < 1000 -- reasonableness checks
      AND ship_charge < 1000
WITH CHECK OPTION
```

Because of EXISTS and other tests, all of which are expected to be successful when retrieving existing rows, this is a most inefficient view for displaying data from **orders**. However, if insertions to **orders** are made only through this view (and you are not already using integrity constraints to constrain data), it is impossible to insert a back-dated order, an invalid customer number, or an excessive shipping weight and shipping charge.

Privileges and Views

When you *create* a view, the database server tests your privileges on the underlying tables and views. When you *use* a view, only your privileges with regard to the view are tested.

Privileges When Creating a View

When you create a view, the database server tests to make sure that you have all the privileges needed to execute the SELECT statement in the view definition. If you do not, the view is not created.

This test ensures that users cannot gain unauthorized access to a table by creating a view on the table and querying the view.

After you create the view, the database server grants you, the creator and owner of the view, at least the Select privilege on it. No automatic grant is made to public as is the case with a newly created table.

The database server tests the view definition to see if the view is modifiable. If it is, the database server grants you the Insert, Delete, and Update privileges on the view, provided that you also have those privileges on the underlying table or view. In other words, if the new view is modifiable, the database server copies your Insert, Delete, and Update privileges from the underlying table or view, and grants them on the new view. If you have only the Insert privilege on the underlying table, you receive only the Insert privilege on the view.

This test ensures that users cannot use a view to gain access to any privileges that they did not already have.

Because you cannot alter or index a view, the Alter and Index privileges are never granted on a view.

Privileges When Using a View

When you attempt to use a view, the database server tests only the privileges you are granted on the view. It does *not* also test your right to access the underlying tables.

If you created the view, your privileges are the ones noted in the preceding paragraph. If you are not the creator, you have the privileges that were granted to you by the creator or someone who had the WITH GRANT OPTION privilege.

The implication of this is that you can create a table and revoke public access to it; then you can grant limited access privileges to the table through views. This can be demonstrated through the previous examples using the **hr_data** table. Its definition is repeated in Figure 11-9.

```
CREATE TABLE hr_data
(
  emp_key INTEGER,
  emp_name CHAR(40),
  hire_date DATE,
  dept_num SMALLINT,
  user-id CHAR(18),
  salary DECIMAL(8,2)
  performance_level CHAR(1)
  performance_notes TEXT
)
```

Figure 11-9 *A table of confidential employee information (duplicate of Figure 11-1)*

In Figure 11-9, the example centered on granting privileges directly on this table. The following examples take a different approach. Assume that when the table was created, the following statement was executed:

```
REVOKE ALL ON hr_data FROM PUBLIC
```

(This is not necessary in an ANSI-compliant database.) Now you create a series of views for different classes of users. For those who should have read-only access to the nonsensitive columns, you create the following view:

```
CREATE VIEW hr_public AS
  SELECT emp_key, emp_name, hire_date, dept_num, user_id
  FROM hr_data
```

Users who are given the Select privilege for this view can see nonsensitive data and update nothing. For the clerks in the Human Resources department who must enter new rows, you create a different view, as shown in the following example:

```
CREATE VIEW hr_enter AS
  SELECT emp_key, emp_name, hire_date, dept_num
  FROM hr_data
```

You grant these users both Select and Insert privileges on this view. Because you, the creator of both the table and the view, have the Insert privilege on the table and the view, you can grant the Insert privilege on the view to others who have no privileges on the table.

On behalf of the clerk in the MIS department who enters or updates new user-ids, you create still another view, as shown in the following example:

```
CREATE VIEW hr_MIS AS
  SELECT emp_key, emp_name, user_id
  FROM hr_data
```

This view differs from the previous view in that it does not expose the department number and date of hire.

Finally, the managers need access to all columns and need the ability to update the performance-review data for their own employees only. These requirements can be met by creating a table, **hr_data**, that contains a department number and a computer user-id for each employee. Let it be a

rule that the managers are members of the departments that they manage. Then the following view restricts managers to rows that reflect only their employees:

```
CREATE VIEW hr_mgr_data AS
  SELECT * FROM hr_data
    WHERE dept_num =
      (SELECT dept_num FROM hr_data
        WHERE user_id = USER)
    AND NOT user_id = USER
```

The final condition is required so that the managers do not have update access to their own row of the table. It is, therefore, safe to grant the Update privilege to managers for this view, but only on selected columns, as shown in this statement:

```
GRANT SELECT, UPDATE (performance_level, performance_notes)
  ON hr_mgr_data TO peter_m
```

Summary

In a database that contains public material or is used only by you and trusted associates, security is not an important consideration and few of the ideas in this chapter are needed. But as more people are allowed to use and modify the data, and as the data becomes increasingly confidential, you must spend more time and be ever more ingenious at controlling the way people can approach the data.

The techniques discussed here can be divided into the following two groups:

- Keeping data confidential

When the database resides in operating system files you can use features of the operating system to deny access to the database. In any case, you control the granting of the Connect privilege to keep people out of the database.

When different classes of users have different degrees of authorization, you must allow them all the Connect privilege. You can use table-level privileges to deny access to confidential tables or columns. Or, you can use a stored procedure to provide limited access to confidential tables or columns. In addition, you can deny all access to tables and allow it only through views that do not expose confidential rows or columns.

- Controlling changes to data and database structure

You safeguard the integrity of the data model by restricting grants of the Resource, Alter, References, and DBA privileges. You ensure that only authorized persons modify the data by controlling the grants of the Delete and Update privileges, and by granting the Update privilege on as few columns as possible. You ensure that consistent, reasonable data is entered by granting the Insert privilege only on views that express logical constraints on the data. Alternatively, you can control the insertion and modification of data, or the modification of the database itself, by limiting access to constrictive stored procedures.

Summary

Understanding Informix Networking

Chapter Overview	3
What Is a Network?	4
Database Management System Configurations	4
A Single-User Configuration	4
Advantages and Disadvantages of a Single-User System	6
A Local Multiuser Configuration	6
Advantages and Disadvantages of Local Multiuser Systems	7
A Remote Configuration	8
Advantages and Disadvantages of Remote Network Connections	8
Single-Machine Configuration Using Network Communication	9
Advantages and Disadvantages of Local Loopback	10
Distributed Databases	10
Advantages and Disadvantages of Distributed Databases	11
Distributed Databases Using Multiple Vendor Servers	11

Connecting to Data on a UNIX Network	12
Example of Client / Server Connections	13
Environment Variables	14
Connection Information	15
SQL Connection Statements	16
Accessing Tables	16
Using Synonyms with Table Names	17
Synonym Chains	18
Protecting Your Data in a Networked Environment	19
Data Protection with INFORMIX-SE	19
Data Protection with INFORMIX-OnLine Dynamic Server	19
Data Replication	19
Archives	20
Data Integrity for Distributed Data	20
Two-Phase Commit	20
Summary	21

Chapter Overview

This chapter gives an overview of the use of databases on a computer network. It introduces some commonly used terminology and illustrates various network configurations. The chapter also presents an overview of how the components of either a local or network connection fit together so that a client application can find data on a database server.

We will discuss the following networking configurations that you can use with Informix databases and point out some of their effects on performance and usability:

- All on one machine
- A simple network connection
- Multiple connections on a network
- Data managed by non-Informix database servers (**INFORMIX-TP/XA**)

It is not enough to simply build a computer network; you have to re-examine how your applications and database servers share data. This chapter also covers the following topics concerned with managing data shared over a network:

- Distributed data
- Connecting to data
- Protecting Your data
- Synonym chains
- Network transparency

The final sections of the chapter discuss protecting your data in a networked environment.

What Is a Network?

A *computer network* is a group of computers, workstations, and other devices connected together over some kind of communications system to share resources and data. A network *site* is simply the location of one of the computers on the network. Sometimes the network sites are widely separated, but they may also be in the same room or building. Two network *sites* can even coexist on the same machine.

To make a practical computer network work, you must master a multitude of technical details regarding hardware and software. There are far too many of these details, and they change too fast to cover them in this book. This chapter gives a conceptual discussion of some of the issues that you will encounter when you use a computer network. For more information, refer to the manual that accompanies the Informix client/server product that you use and/or to a textbook on computer networks.

Database Management System Configurations

A *relational database management system* (RDBMS) includes all of the components necessary to create and maintain a relational database. An Informix RDBMS has several pieces: the user interface, the database server and the data itself. It is easiest to think of these pieces as all being located in the same computer, but many other arrangements are possible. In some arrangements, the application and the database server are on different computers, and the data is distributed across several others.

The client applications of an RDBMS do not need to be modified to run on a network. The communications tools handle the tasks of locating and attaching to the database servers. To the application, a database on a networked computer appears no different than a database on the computer where the application resides.

A Single-User Configuration

Figure 12-1 shows a diagram of a simple database management system on a single computer. The organization in Figure 12-1 is one you would typically find on a personal computer running the DOS operating system. It is unusual to have a single-user situation on a UNIX system, but you certainly can. One example of a single-user UNIX system would be a desktop workstation in a development environment.

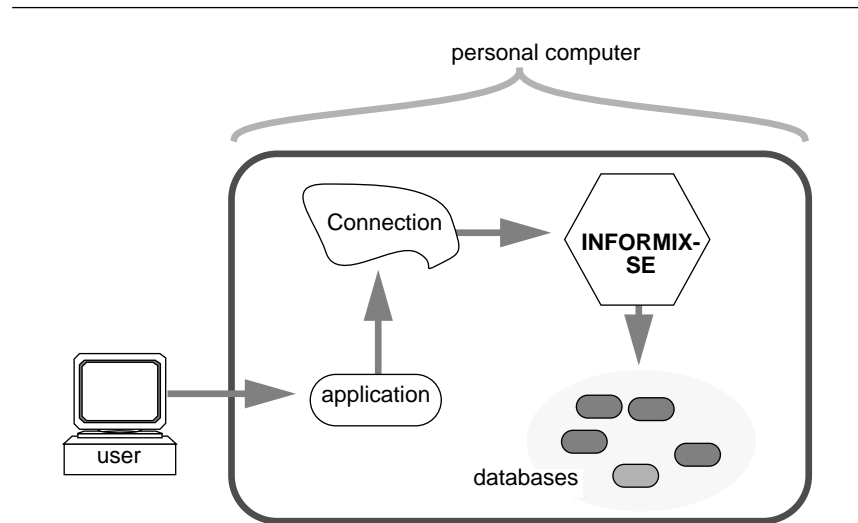


Figure 12-1 A database management system on a personal computer

The components of the system in Figure 12-1 are found in the following list:

- An application program
Any program that issues a query can be the application. It could, for example, be a program written in **INFORMIX-4GL**, a C language program with embedded SQL, or compiled screen forms and reports.
- A connection
On a simple system such as this one, the communication component is frequently so well integrated with the system that it is usually omitted from diagrams and not discussed. However, it does exist.
- A database server
The database server receives queries from the application, searches the database, and returns information to the application. The database server manages or administers the databases that it controls.
Because Figure 12-1 illustrates a personal computer, the database server must be **INFORMIX-SE**. The database server in this diagram is a *local* server because it resides on the same host computer as the client application.

- **Databases.** The databases are usually stored on a magnetic disk.
If this system were a UNIX system with **INFORMIX-OnLine Dynamic Server**, the database might be located on some other medium, such as a WORM (write-once read-many-times) drive controlled by **INFORMIX-OnLine/Optical**.

Advantages and Disadvantages of a Single-User System

A configuration involving only one computer is the easiest configuration to set up and maintain, and it gives the fastest access to data. However, the data is not available to users on other computers and the size of the databases or the amount of processing needed may outgrow the capacity of the computer.

A Local Multiuser Configuration

Figure 12-2 shows another database management system configuration, a multiuser system with a local database server such as one you might find on a computer with a UNIX operating system.

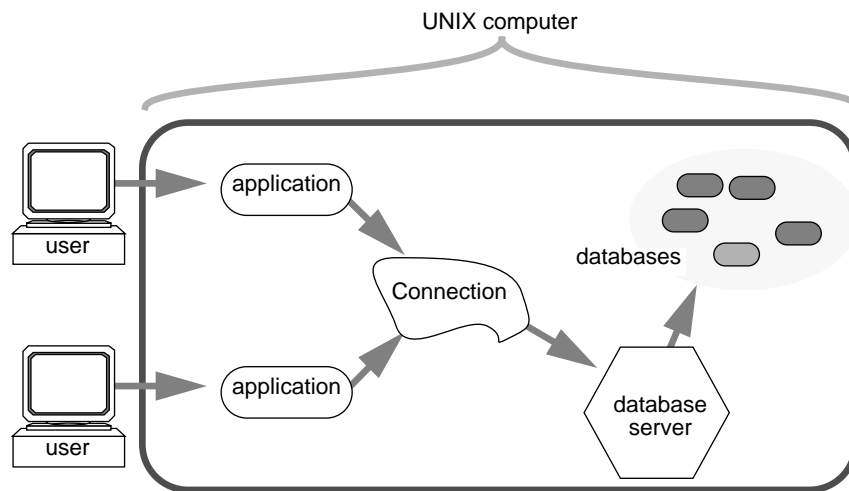


Figure 12-2 A database management system on a UNIX computer

The components of the systems in Figure 12-2 are similar to the components in Figure 12-1, as described in the following list:

- **Application programs**
Two or more applications use the same database server to access information in the databases. You might have two users at individual terminals, as shown, or you might have multiple windows on a single workstation.
- **A connection**
On a local UNIX system, the following types of connection are possible:
 - Inter-process communication (IPC)
 - Network connectionIPC is a UNIX feature that transfers information very quickly between the application and the database server. It is available only when the application and the database server reside on the same computer. **INFORMIX-SE** databases use a type of IPC connection called *unnamed pipes* and **INFORMIX-OnLine Dynamic Server** use an IPC connection technique called *shared memory*.
- **A database server, either **INFORMIX-OnLine Dynamic Server** or **INFORMIX-SE****
- **Databases**

Advantages and Disadvantages of Local Multiuser Systems

A configuration that allows multiple users gives better access to the data than does a single-user system. With a local database, it is still easy to set up and maintain. However, like the single-user system, the data is not available to users on other computers and the size of the databases or the amount of processing needed may outgrow the capacity of the computer.

IPC shared memory provides very fast communication between the client application and the database server. However, IPC shared memory communication is vulnerable to programming errors if the client application does explicit memory addressing or overindexes data arrays. Such errors do not affect the application if you use IPC unnamed pipes or network communication. (See “Single-Machine Configuration Using Network Communication” on page 12-9.)

A Remote Configuration

Figure 12-3 shows a *remote* configuration, where the application resides on one computer, and the database server and its associated databases reside on another computer on the network. In contrast, the database servers in Figure 12-1 and Figure 12-2 are local database servers.

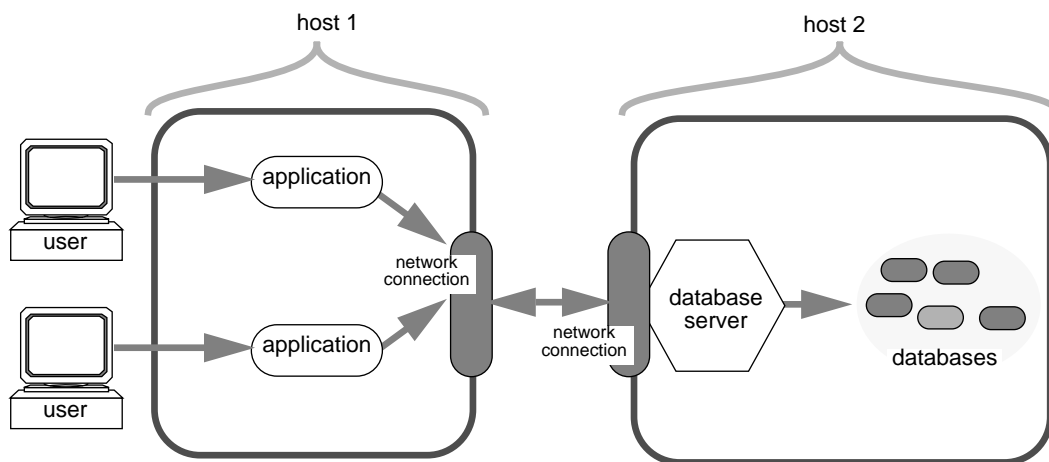


Figure 12-3 A simple network connection

In Figure 12-3 the applications might be **INFORMIX-ESQL/COBOL** or **INFORMIX-ESQL/C** applications. The database server might be **INFORMIX-OnLine Dynamic Server (UNIX)** or **INFORMIX-OnLine for NetWare (PC)**.

Different computers are referred to as sites or *host machines*. The database server in Figure 12-3 is a *remote* database server because it is on a different host machine, or site, from the application that uses its services. A database server can be local with respect to one application and remote with respect to another application, as illustrated in Figure 12-6 on page 12-13.

Advantages and Disadvantages of Remote Network Connections

The configuration shown in Figure 12-3 is an example of *distributed processing*. In distributed processing, multiple computers contribute to a single computing task. In this example, *host1* handles the computing requirements of the application, such as screen display, generation of reports and printing, and

host2 handles the computing required to manipulate information in the databases. Using a network also gives the client application the opportunity to access data from several computers.

Response from a database server using network communication is not as fast as response from a database server using local IPC communication, because of the extra computing required to prepare the data for network communication and because of the transmit time. A network is somewhat more difficult to configure and maintain than a local system that uses IPC communication.

Single-Machine Configuration Using Network Communication

Figure 12-4 shows a configuration that behaves as if there were multiple sites residing on one machine. In this configuration, which is known as *local loopback*, all the components are on the same computer, but the connections are made as if they were connected through a network.

Because the connections use network software, the database server appears to the application as a remote site. The dashed line indicates the division between the two *sites*. The database administrator configures the system to provide local or local loopback connections (or both).

The software for the client application and the software for the database server can be stored in the same directory, or the software can be stored in two separate directories.

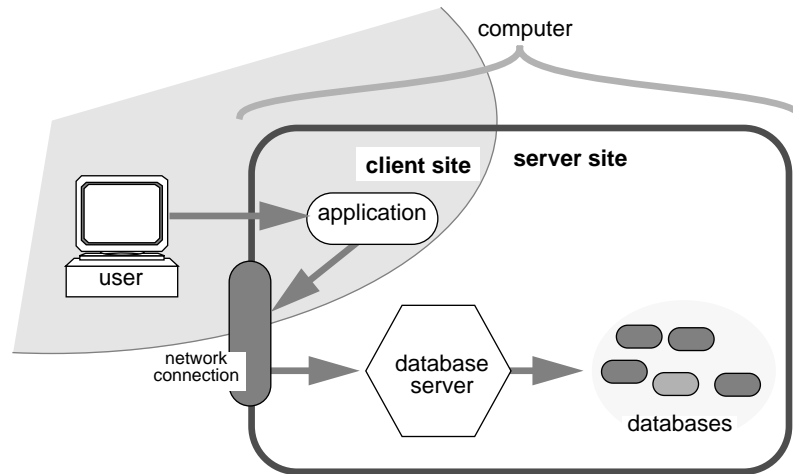


Figure 12-4 An example of local loopback

Advantages and Disadvantages of Local Loopback

Local loopback allows you to test network operations without requiring an actual remote machine, but it is not as fast as a local IPC connection. However, unlike IPC shared-memory communication, local loopback is not vulnerable to corruption due to memory-addressing errors or overindexed data arrays. (See “Advantages and Disadvantages of Local Multiuser Systems” on page 12-7.)

Distributed Databases

Although a network lets you separate the application from the data, the application still is limited to the contents of a single database. With most database servers, you only can query or modify tables in the current database.

A *distributed database* has information on multiple databases organized so that it appears as a single database to the user. The data can be maintained by a variety of database servers and located on computers supported by different operating systems and communication networks.

The **INFORMIX-OnLine Dynamic Server** database server allows you to query data in multiple databases anywhere on the network. When the **INFORMIX-TP/XA** feature is added to **INFORMIX-OnLine Dynamic Server**, you can create global transactions that span multiple computer systems and even multiple XA-compliant database systems from different vendors. **INFORMIX-SE** does not provide for distributed database processing.

Advantages and Disadvantages of Distributed Databases

You may find distributed databases useful because operations that use databases are often naturally distributed into separate pieces, either organizationally, geographically, or both. A distributed database system includes the following advantages:

- Local data can be kept locally where it is most easily maintained and most frequently used.
- Data from remote sites is available to all users.
- Duplicate copies can be maintained for safety of the data.

A distributed database system has the following disadvantages:

- Management of the distributed system is more involved than management of a single-host system.
- Network access is slower than local access.

Distributed Databases Using Multiple Vendor Servers

You need **INFORMIX-TP/XA** if you use database management systems from multiple vendors to store your data and you want to use one application to access the data. **INFORMIX-TP/XA** is a library of functions that allows the **INFORMIX-OnLine Dynamic Server** database server to act as a resource manager in a distributed transaction-processing (DTP) environment that follows an interface standard which was defined by the X/Open Company.

The terminology used by the XA standard is different from the terminology used by Informix products and documentation. A *transaction manager* acts as an intermediary and relays requests from a user interface to a *resource manager*. Transaction managers are third-party products such as TUXEDO. In this context, a resource manager corresponds to a database server. Figure 12-5 illustrates a configuration that uses transaction processing.

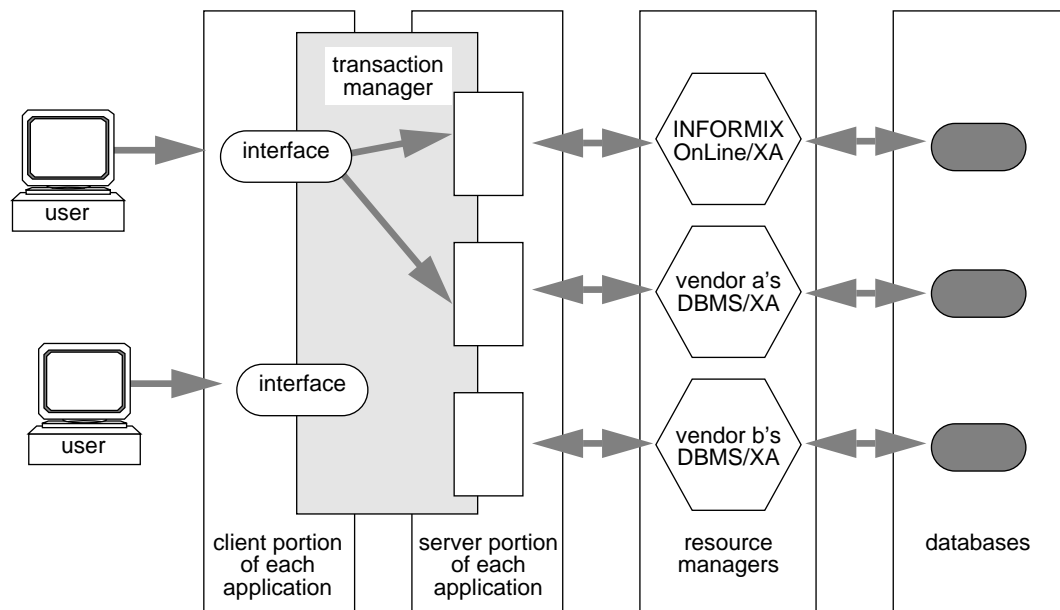


Figure 12-5 A configuration using a transaction manager

Connecting to Data on a UNIX Network

When the application and data are moved to separate computers, two questions immediately arise: What connections can you implement? How do you instruct your applications to find the data that is now somewhere else?

In fact, connecting a client application to a database server that is on a networked computer is no different from connecting to a local database server. To connect a client application with a database, you must consider the following parts of your configuration:

- Environment variables
- Connection information
- Connection statements

This section summarizes how connections are made for Version 6.0 clients and servers. Detailed instructions for setting up local and network connections for both Version 6.0 products and earlier products are given in the *INFORMIX-OnLine Dynamic Server Administrator's Guide* and the *INFORMIX-SE Administrator's Guide*.

Example of Client / Server Connections

Figure 12-6 shows both local and networked connections of Version 6.0 products. Although all these connections are possible, you cannot do them all at once. Client X can make connection 1, 2, or 3, and Client Y can make connection 4 or 5.

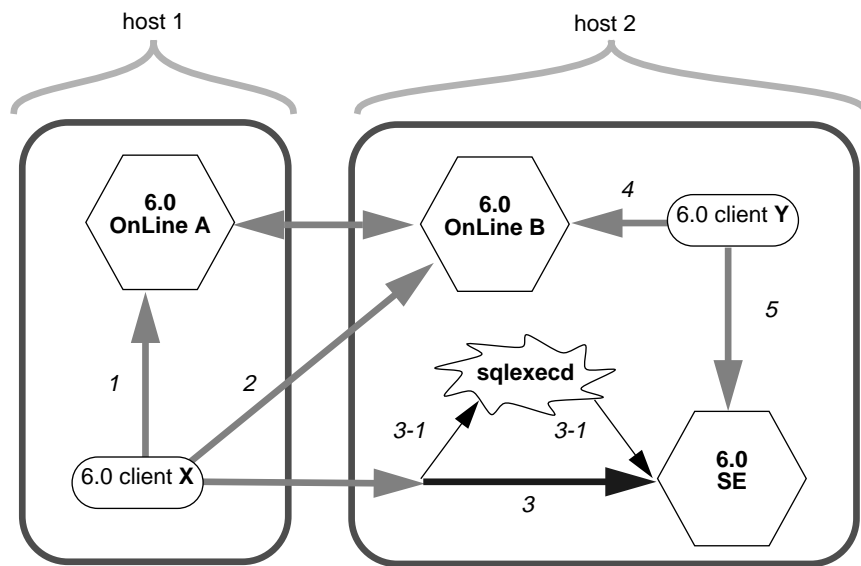


Figure 12-6 Local and network connection of version 6.0 products

You can connect clients directly to **INFORMIX-OnLine Dynamic Server** database servers using either shared memory (IPC) or a network communications. Notice also that **OnLine** database servers (**OnLine A** and **OnLine B**) can communicate with each other.

Local clients can connect directly to an **INFORMIX-SE** database server using unnamed pipes (IPC). Clients using network communications connect to **INFORMIX-SE** by connecting first to a *daemon*, **sqlxecd**. (A daemon is a background process that listens for requests from other programs.) The **sqlxecd** daemon is started by the system administrator.) When a client requests a connection to an **INFORMIX-SE** database server, **sqlxecd** notices the request and creates a temporary (light gray arrows) connection between the client and **SE**. This temporary connection (for example, 3-1) enables the client and **INFORMIX-SE** database server to establish a direct connection (3). Then **sqlxecd** removes itself from the loop, leaving the client attached to the **SE** database server.

Environment Variables

The Informix administrator must make sure that each user sets the correct environment variables. The following list shows the most important environment variables used by **INFORMIX-OnLine Dynamic Server** and **INFORMIX-SE**:

- PATH
- INFORMIXDIR
- INFORMIXSERVER
- TERM
- DBPATH

The **INFORMIXDIR** environment variable must be set to the full pathname of the directory where the Informix files reside. The **PATH** environment variable must include the full pathname of the directory where the executables for **INFORMIX-SE** and/or **INFORMIX-OnLine Dynamic Server** reside. These two environment variables are required. After the Informix administrator has specified them, you usually do not change them.

The **INFORMIXSERVER** environment variable is the name of the default database server. It is also a required variable. You may choose to change **INFORMIXSERVER** when you change applications.

The **TERM** (or **TERMCAP** and/or **INFORMIXTERM**) environment variable enables the clients to recognize and communicate with your terminal. These variables are system (and terminal) dependent. To set them, you may need the assistance of your system administrator.

The **DBPATH** environment variable is optional. If the application does not fully specify the location of an **SE** database server, the directories listed in **DBPATH** are searched by the database server to find the specified database.

The DBPATH environment variable is also used to specify directory names for reports, forms, and command files used by both **INFORMIX-OnLine Dynamic Server** and **INFORMIX-SE**.

These environment variables are described in detail in Chapter 4 of the *Informix Guide to SQL: Reference*.

Connection Information

When the **INFORMIX-OnLine Dynamic Server** or **INFORMIX-SE** administrator installs and configures a database server, he or she must cooperate with the UNIX system administrator to specify the information that allows a client application to locate a database server. The application specifies a database server and the Informix connectivity tools make the connection. The connection is transparent; the client does not need to know where the database server resides.

The connection information is specified in the **\$INFORMIXDIR/etc/sqlhosts** file and two UNIX systems files, **/etc/hosts** and **/etc/services**. The three files, working together, specify the location of the database server and the type of connection (protocol) for the database server. (See Figure 12-7.) On each computer, every database server that might be accessed by an application must have an entry in the **\$INFORMIXDIR/etc/sqlhosts** file and corresponding entries in the **/etc/hosts** and **/etc/services** files. These files are described in detail in the *INFORMIX-OnLine Dynamic Server Administrator's Guide* and *INFORMIX-SE Administrator's Guide*.

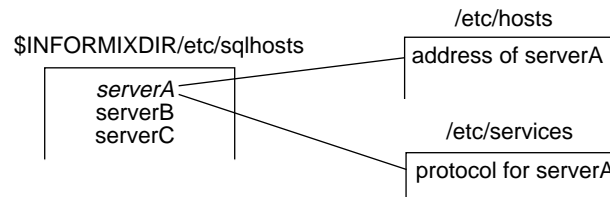


Figure 12-7 **Connectivity Information for Networks**

SQL Connection Statements

Before it can do anything else, the client application must *open* a database by executing a CONNECT statement or a DATABASE statement. The CONNECT statement is preferred because it conforms with both ANSI and X/Open standards that attempt to specify uniform syntax for networked and non-networked environments.

Here is an example of the CONNECT statement:

```
CONNECT TO databasename@servername
```

If the *servername* is omitted from the statement, the application connects to the default database server specified by the INFORMIXSERVER environment variable.

If the *databasename* is omitted, the application connects to the database server *servername* but does not open a database. Before you can use a database, you must issue a DATABASE, CREATE DATABASE, or START DATABASE statement.

Notice that the CONNECT statement does not give any information about the location of the database server. The location information for the database server is in the **sqlhosts** file. (Refer to “Connection Information” on page 12-15.)

The complete syntax for the CONNECT and DATABASE statements is covered in Chapter 1 of the *Informix Guide to SQL: Syntax*.

Accessing Tables

The database that is opened by a CONNECT, DATABASE or CREATE DATABASE statement is the *current* database. If you are using **INFORMIX-OnLine Dynamic Server**, you can query tables that are not in the current database. To refer to a table in a database other than the current database, include the database name as part of the table name, as illustrated in the following SELECT statement:

```
SELECT name, number FROM salesdb:contacts
```

The database is **salesdb**. The table in **salesdb** is named **contacts**. You can use the same notation in a join. When you must specify the database name explicitly, the long table names can become cumbersome unless you use aliases to shorten them. This is shown in the following example:

```
SELECT C.custname, S.phone
      FROM salesdb:contacts C, stores:customer S
      WHERE C.custname = S.company
```

To specify a table in a database that is managed by a different **INFORMIX-OnLine Dynamic Server** database server, you must qualify the database name with a *database server name*. For example, the following SELECT statement refers to table **customer** from database **masterdb**, which resides on the database server **central**:

```
SELECT O.order_num, C.fname, C.lname
      FROM masterdb@central:customer C, sales@boston:orders O
      WHERE C.customer_num = O.Customer_num
      INTO TEMP mycopy
```

In the example, two tables are being joined. The joined rows are stored in a temporary table in the current database. The tables are located in two database servers; **central** and **boston**. Informix allows you to *over-qualify* (to give more information than is required) table names, so it is possible, in this example, that the current database is **masterdb** or **sales**.

Using Synonyms with Table Names

A *synonym* is a name that you can use in place of another name. The main use of the CREATE SYNONYM statement is to make it more convenient to refer to tables that are not in the current database.

The preceding example has been revised, as shown in the following example, to use synonyms for the table names:

```
CREATE SYNONYM mcust FOR masterdb@central:customer;

CREATE SYNONYM bords FOR sales@boston:orders;

SELECT bords.order_num, mcust.fname, mcust.lname
      FROM mcust, bords
      WHERE mcust.customer_num = bords.Customer_num
      INTO TEMP mycopy
```

The CREATE SYNONYM statement stores the synonym name in the system catalog table **syssyn**table in the current database. The synonym is available to any query made within that database.

A short synonym makes it easier to write queries, but synonyms can play another role. They allow you to move a table to a different database, or even to a different computer, while leaving your queries the same.

Suppose you have several queries that refer to the tables **customer** and **orders**. The queries are embedded in programs, forms, and reports. The tables are part of database **stores6**, which is kept on database server **avignon**.

Now the decision is made that the same programs, forms, and reports are to be made available to users of a different computer on the network (database server **nantes**). Those users have a database that contains a table named **orders** containing the orders at their location, but they need access to the table **customer** at **avignon**.

To those users, the **customer** table is external. Does this mean you must prepare special versions of the programs and reports, versions in which the **customer** table is qualified with a database server name? A better solution is to create a synonym in the users' database, as shown in the following example:

```
DATABASE stores6@nantes;  
CREATE SYNONYM customer FOR stores6@avignon:customer
```

When the stored queries are executed in your database, the name **customer** refers to the actual table. When they are executed in the other database, the name is translated through the synonym into a reference to the external table.

Synonym Chains

To continue the preceding example, suppose that a new computer is added to your network. Its name is **db_crunch**. The **customer** table and other tables are moved to it to reduce the load on **avignon**. You can reproduce the table on the new database server easily enough, but how can you redirect all accesses to it? One way is to install a synonym to replace the old table, as shown in the following example:

```
DATABASE stores6@avignon EXCLUSIVE;  
RENAME TABLE customer TO old_cust;  
CREATE SYNONYM customer FOR stores6@db_crunch:customer;  
CLOSE DATABASE
```

When you execute a query within **stores6@avignon**, a reference to table **customer** finds the synonym and is redirected to the version on the new computer. This is also true of queries executed from database server **nantes**. The synonym in the database **stores6@nantes** still redirects references to **customer** to database **stores6@avignon**; however, the new synonym there sends the query to database **stores6@db_crunch**.

Chains of synonyms can be useful when, as in this example, you want to redirect all access to a table in one operation. However, you should update all users' databases as soon as possible so their synonyms point directly to the table. You incur extra overhead in handling the extra synonyms and, if any computer in the chain is down, the table cannot be found.

It is possible to run an application against a local database and later to run the identical application against a database in another machine. The program runs equally well in either case (although it may run more slowly on the network database). As long as the data model is the same, a program cannot tell the difference between a local database server and a remote one.

Protecting Your Data in a Networked Environment

This section gives an overview of data-protection features used on a network. Chapter 4, "Modifying Data," has a general discussion of data protection.

Data Protection with INFORMIX-SE

INFORMIX-SE databases use the normal UNIX file structures, so **SE** databases can be archived with the usual operating system backup procedures. On a network, you can write the archives to a device in another location.

Data Protection with INFORMIX-OnLine Dynamic Server

OnLine includes several different tools that make multiple copies of data. Each one has its own unique task in the area of data protection. The following discussions give a general description of these tools and highlight the distinctions between them.

Data Replication

In a general sense, *data replication* means that a given piece of data has several distinct representations on several distinct servers. **OnLine** does data replication by using two networked computers. Each computer has an **OnLine**

database server and databases that have identical characteristics. One database server is the *primary* and the other is the *secondary*. Data is always written to the primary server and then transferred to the secondary server.

Applications can *read* data from either database server. Thus the secondary site can provide a dual purpose: It provides data protection and it provides improved performance for users at the secondary site who need to read, but not write, data.

If the primary server (call it serverA) fails for any reason, the secondary server (call it serverB) can become an independent server. Users who would normally use the primary (serverA) can be switched to serverB. Service to all the users can continue with a minimum of disruption while the original primary server (serverA) is being repaired.

Data replication provides good data protection and continuity of service, but it is expensive: Memory must be provided for two complete copies of the data, shipping the data to a remote site affects performance, and management requires the attention of the **OnLine** administrators on both sites.

Archives

OnLine provides specialized tools for making archives. Archives can be prepared locally or on a networked computer. Archives should be stored in a location that is physically removed from the database server, so it may be convenient to build the archives on a networked computer located at a different physical site.

Data Integrity for Distributed Data

INFORMIX-OnLine Dynamic Server allows you to update data in several databases on different database servers. For example, a purchase order (a single transaction) might require updating of information in databases on different database servers. To maintain the integrity of the data, it is essential that either all the different databases are updated or none of the databases are updated.

Two-Phase Commit

Two-phase commit is a protocol that coordinates work performed at multiple database servers on behalf of a single transaction. Unlike the data-replication and archiving tools discussed earlier in this section, two-phase commit *does not* make two copies of the data. It protects the validity of *one* transaction,

which involves several different databases that may or may not be on a network. Because two-phase commit involves only one transaction, it is not usually classed as data protection.

A transaction that involves multiple database servers is called a *global transaction*. Two-phase commit is a natural extension of transaction handling, which is discussed in “Interrupted Modifications” on page 4-22. The *INFORMIX-OnLine Dynamic Server Administrator’s Guide* discusses two-phase commit in detail.

The two-phase commit begins when a user process has completed all of its work and requests a commit of the global transaction.

Phase 1

The current database server asks each participating database server if it can commit its local transactions. Each database server responds “Yes” or “No.”

Phase 2

If all the database servers respond affirmatively, the current database server tells each one to commit its transactions and then the global transaction is complete. If *any* database server responds negatively or does not respond, all database servers are instructed to abort the local transactions.

Summary

A network allows a client application to run in one machine, while the database server operates in another computer to which the data is physically attached. This provides distributed processing and the possibility of distributed database access. Many possible combinations exist of network software, operating systems, and database servers, and each has subtleties that must be mastered.

Summary



Section III: Using Advanced SQL





Optimizing Your Queries

Chapter Overview	3
Optimizing Techniques	4
Verifying the Problem	4
Considering the Total System	5
Understanding the Application	5
Measuring the Application	6
Manual Timing	6
Time from Operating System Commands	6
Time from the Programming Language	6
Finding the Guilty Functions	7
Keeping an Open Mind	7
The Query Optimizer	8
How the Optimizer Works	8
How the Optimizer Joins Tables	8
Obtaining Input	9
Assessing Filters	10
Selecting Table-Access Paths	11
Selecting the Query Plan	11
Reading the Plan	12
Time Costs of a Query	14
Activities in Memory	14
Disk-Access Management	14
Page Buffers	16
The Cost of Reading a Row	16
The Cost of Sequential Access	17
The Cost of Nonsequential Access	18
The Cost of Rowid Access	18
The Cost of Indexed Access	18

The Cost of Small Tables	19
The Cost of NLS Functionality	19
The Cost of Network Access	20
The Importance of Table Order	22
A Join Without Filters	22
A Join with Column Filters	23
Using Indexes	26
The Sort-Merge Join Technique	27
Making Queries Faster	27
Preparing a Test Environment	28
Studying the Data Model	28
Studying the Query Plan	29
Examining the Indexes	29
Replacing Autoindexes with Indexes	29
Creating Data Distributions on Filtered Columns	29
Using Composite Indexes	31
Using oncheck on Suspect Indexes	31
Dropping and Rebuilding Indexes After Updates	31
Refining the Column Filters	31
Avoiding Correlated Subqueries	32
Avoiding Difficult Regular Expressions	32
Avoiding Noninitial Substrings	33
Rethinking the Query	33
Rewriting Joins Through Views	33
Avoiding or Simplifying Sorts	33
Eliminating Sequential Access to Large Tables	34
Using Unions to Avoid Sequential Access	35
Reviewing the Optimization Level	36
Using a Temporary Table to Speed Queries	36
Using a Temporary Table to Avoid Multiple Sorts	36
Substituting Sorting for Nonsequential Access	37
Summary	41

Chapter Overview

How much time should a query take? How many disk operations should the computer perform while executing a query? For many queries, it does not matter as long as the computer finds the information faster than a human can. But some queries must be performed in a limited amount of time or must use only a limited amount of machine power.

This chapter reviews techniques for making queries more efficient. *It assumes that you work with an existing database* and cannot change its arrangement of tables. (Techniques of designing a new database for reliability and performance are discussed in Chapters 8 through 11.)

This chapter covers the following topics:

- A discussion of techniques for optimizing software, emphasizing all the things to look at before you change any SQL statement.
- A discussion of the *optimizer*, the part of the database server that decides how to perform a query. When you know what the optimizer looks for when it forms a query plan, you can help it form more efficient ones.
- A discussion of the operations that take time during a query, so you can better choose between fast and slow operations.
- An assortment of techniques to help the optimizer choose the fastest way to accomplish a query.

This chapter concentrates on SQL performance, but performance problems can arise from other parts of the programs in which SQL statements are embedded. Two books that address general issues of performance are *The Elements of Programming Style* by Kernighan and Ritchie (McGraw-Hill, 1978) and *Writing Efficient Programs* by Jon Louis Bentley (Prentice-Hall, 1982). Much of Bentley's advice can be applied to SQL statements and database design.

Optimizing Techniques

Before you begin optimizing your SQL statements, begin considering them as part of a larger system that includes the following components:

- One or more *programs*
The saved queries, compiled screen forms and reports, and programs in one or more languages in which SQL statements are embedded
- One or more *stored procedures*
The compiled procedures, comprising SQL and SPL statements, which are stored in an executable form in the database
- One or more *computers*
The machines that store the programs and the databases
- One or more *maintainers*
The people responsible for maintaining the programs
- One or more *users*
The people whose work the system is supposed to amplify or simplify
- One or more *organizations*
The groups that own the computers and choose what work is to be done

You may work in a large corporation where each of these components is separate, or you may be the proprietor, maintainer, and sole user of a desktop workstation. In every case, it is important to recognize two points about the system as a whole. First, the goal of the system is to serve its *users*. Any effort you spend optimizing programs is wasted unless the result helps the users in some way. Second, SQL statements are only a small part of the system: Effective improvements can be found in many parts of a system.

The following paragraphs outline a general procedure for analyzing any computer-performance problem. Follow the procedure to help avoid overlooking possible solutions, including the nontechnical ones that sometimes provide the best answer to performance problems.

Verifying the Problem

Before you begin optimizing SQL statements, be sure the problem lies with the program. How much more effective can the user be if you make the program faster? If the answer is “not much,” look elsewhere for a solution.

Considering the Total System

Consider the entire system of programs, computers, and users within an organization. You might find a better solution by changing schedules or managing resources differently. If you are using **INFORMIX-OnLine Dynamic Server**, you might want to make sure that your system is tuned for efficiency. Maybe the problem operation would be faster if it were done at a different time, on a different machine, or with different jobs running at the same time; or maybe not. But you need to consider these possibilities before you decide the best solution.

For information on tuning your **INFORMIX-OnLine Dynamic Server** system, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

Understanding the Application

Learn all about the application as a whole. A database application usually has many parts, such as saved queries, screen forms, report specifications, and programs. Consider them together with the user's procedures, and ask yourself the following questions:

- *What* is being done?
You may identify excess or redundant steps that can be eliminated. Even if every step or action is important, it can help you to know them all and their sequence of use.
- *Why* is it being done?
Especially in older applications, you may find steps that serve no purpose; for example, an operation might have been put in long ago for debugging and never removed.
- *For whom* is it being done?
Make sure that all output from the application is wanted and used by someone. You may find output that nobody wants any more, or the output may be needed infrequently or in a simpler format.
- *Where* is the slow part?
Isolate as closely as possible which steps are too slow. Your time is limited too; spend time where it yields the best results.

By taking the time to understand the application, you can know all its parts, how they are used, whether they are all essential, and which parts are too slow.

Measuring the Application

You cannot make meaningful progress on a performance problem until you have measured it. You must find a way to take repeatable, quantitative measurements of the slow parts of the application. This is crucial for the following reasons:

- Without numbers you cannot accurately and specifically describe the problem to users or to the organization.
To convey the problem appropriately, you need to determine measurements such as “The report runs in 2 hours, 38 minutes,” or “The average update takes 13 seconds, but during peak hours it takes as long as 49 seconds.”
- Without numbers you cannot set meaningful goals.
When you have specific measurements, you can obtain agreement on numeric goals for performance.
- Without numbers you cannot measure and demonstrate your progress.
You need measurements to detect and demonstrate small improvements and to choose between alternative solutions.

Manual Timing

You can obtain repeatable measurements with a hand-operated stopwatch. With practice, most people can obtain timings that are repeatable within two-tenths of a second. Manual timing is useful if you are measuring only a few events that are at least several seconds long. (Very short events should be measured over several cycles of averaging to reduce the error in measurement.)

Time from Operating System Commands

Your operating system probably has a command that displays the time. You can package an operation to be timed between two time commands in a command script.

Time from the Programming Language

Most programming languages have a library function for the time of day. If you have access to the source code, you can insert statements to measure the time of specific actions. For example, if the application is written in

INFORMIX-4GL, you can use the **CURRENT** function to obtain the current time as a **DATETIME** value. A **4GL** program can perform automated timing with code similar to that shown in the following example:

```
DEFINE start_time DATETIME HOUR TO FRACTION(2),
      elapsed INTERVAL MINUTE(4) TO FRACTION(2)
LET start_time = EXTEND(CURRENT,HOUR TO FRACTION(2))

      { -- here perform the operation to be timed -- }

LET elapsed = EXTEND(CURRENT,HOUR TO FRACTION(2))-start_time
DISPLAY 'Elapsed time was ',elapsed
```

Elapsed time, in a multiprogramming system or network environment where resources are shared among multiple processes, does not always correspond to execution time. Most C libraries contain a function that returns the CPU time of a program, and C functions can be called from **4GL** programs and **ACE** reports.

Finding the Guilty Functions

In most programs, a very small fraction of the code (typically 20 percent or less) accounts for the bulk of the program execution time (typically 80 percent or more). The *80-20 rule*, as it is called, holds in most cases; often, the proportions are even greater. After you establish a timing mechanism, use it to define the *hot spots* in the application, that is, the statements that consume most of the time. In addition, many computer systems provide profiling tools that can help determine the location of hot spots.

Keeping an Open Mind

With embedded SQL, in which one statement can trigger thousands of disk accesses, it is possible that the guilty 20 percent includes some SQL. However, *this is by no means certain*. If you do not begin your examination of the problem with an open mind, it is easy to overlook the part of a program that is consuming the most time.

If the sluggish operations are not due to the SQL, turn to the books cited on page 13-3. If the SQL statements need attention, you need to understand how to structure them so they run more efficiently.

The Query Optimizer

The *optimizer* component of the database server decides how to perform a query. Its most important job is to decide the order in which to examine each table row. To make that decision, it has to decide on the most efficient way to access each table—by a sequential scan of its rows, by an existing index, when joining tables by a temporary index built for the occasion, or by sorting—and it has to estimate how many rows each table contributes to the final result.

Optimizer design is not a science, and the optimizer is one part of a database server that is under continuous development and improvement.

How the Optimizer Works

The optimizer in Informix database servers formulates all possible query plans. For each plan, it estimates the number of table rows to be examined, disk pages to be read, and the network accesses that the plan requires. It selects the plan with the lowest estimate. The following sections provide a general overview of some of the steps that the optimizer takes to evaluate queries. These sections are meant to give you a basic idea of what the optimizer does so that you can understand how to streamline your queries, as described in later sections.

How the Optimizer Joins Tables

The optimizer in Informix database servers does an exhaustive search on all joined tables, regardless of whether the joins contain filters or indexes and removes redundant pairs as it builds the paths. Consider the following example:

```
SELECT ... FROM customer c, items i, orders o
      WHERE c.customer_num = o.customer_num
      AND o.order_num = i.order_num
```

The optimizer forms all possible join pairs between the tables and then removes redundant join pairs. It keeps the least-expensive join triples as shown in the following example.

```
(c,i) (c,o) (i,c) (i,o) (o,c) (o,i)
```

Obtaining Input

The optimizer can be successful only if its estimates are accurate. (They need not be accurate in absolute terms, but they must be relatively accurate so that better plans produce lower estimates than worse plans.) However, the optimizer has only limited information. To make its work take a small fraction of the execution time, it has to make do with the information in the system catalog tables and other summary information. No time exists, for example, for the optimizer to perform a `SELECT COUNT(*)` operation to obtain an accurate count of the rows in a table.

The information available to the optimizer comes from the system catalog tables. (See Chapter 2 of the *Informix Guide to SQL: Reference* for information on system catalog tables.) In all Informix database servers, this information includes

- The number of rows in a table (as of the most recent `UPDATE STATISTICS` command)
- Whether a column is constrained to be unique
- The indexes that exist on a table, including the columns they encompass, whether they are ascending or descending, and whether they are clustered
- Data distribution information about filtered columns

The system catalog tables maintained by **INFORMIX-OnLine Dynamic Server** supply the following additional input:

- The number of disk pages occupied by row data
- The depth of the index B+ tree structure (a measure of the amount of work needed to perform an index lookup)
- The number of disk pages occupied by index entries
- The number of unique entries in an index (divided by the number of rows, this suggests how many rows might match to a given key)
- Second-largest and second-smallest key values in an indexed column

Only the second-largest and second-smallest key values are noted because the extreme values might be special out-of-range signals. The database server assumes that key values are distributed smoothly between the second largest and second smallest. Only the initial four bytes of these keys are stored. In addition, if the index columns have distributions, the distributions, rather than the key value, can be used to estimate the number of rows that match a filter.

Assessing Filters

The optimizer first examines the expressions in the WHERE clause by looking for filters. The optimizer estimates the *selectivity* of each filter it finds. The selectivity is a number between 0 and 1 that indicates the fraction of rows the optimizer thinks the filter can pass. A very selective filter that passes very few rows is assigned a selectivity near 0; a filter that passes most rows is assigned a selectivity near 1. For details on this process, see “Filter Selectivity Assignments” on page 13-30.

The optimizer also notes other information about the query, such as what filter columns are indexed and whether an index is sufficient. If only the values in an index are required (**INFORMIX-OnLine Dynamic Server**), then no table read can be required (it is faster to read the index pages without the data pages).

In addition, the optimizer notes whether an index can be used to evaluate a filter. For this purpose, an indexed column is a column that has an index, or one that is named first in a composite index. Consider the following cases:

- When an indexed column is compared to a literal, a host variable, or an uncorrelated subquery, the database server can look up matching values in the index instead of reading the rows.
- When an indexed column is compared to a column in another table (a join expression), the database server can use the index to find matching values, provided that the query plan calls for reading rows from the other table first. The following join expression is an example:

```
WHERE customer.customer_num = orders.customer_num
```

If rows of **customer** are read first, values of **customer_num** can be looked up in an index on **orders.customer_num**.

- Whether an index can be used in processing an ORDER BY clause
If all the columns in the clause appear in one index in the same sequence, the database server can use the index to read the rows in their ordered sequence, thus avoiding a sort.
- Whether an index can be used in processing a GROUP BY clause
If all the columns in the clause appear in one index in the same sequence, the database server can read groups with equal keys from the index without needing a sort.

Selecting Table-Access Paths

The optimizer next chooses what it estimates to be the most efficient way of accessing each table named in the query. It has the following choices:

- To read the rows of the table sequentially
- To read one of the indexes for the table and read the rows to which the index points
- To create and use a temporary index
- To perform a sort merge

The choice between the first two options depends in large part on the presence of a filter expression. When any filter exists on an indexed column, the database server chooses the selected rows through the index and processes only those rows.

When no filter exists, the database server must read all the rows anyway. It is usually faster to simply read the rows, rather than reading the pages of an index and then reading the rows. However, if the rows are required in sorted order, there may be a net savings in reading the index and through it reading the rows in sorted order.

The optimizer might choose a third option, creating a temporary index, in two cases. When neither table in a join has an index on the joining column, and the tables are large enough, the optimizer may decide that it is quicker to build an index to one table than to read through that table sequentially for each row of the other table. You can also use a temporary index to generate rows in sorted or grouped sequence. (The alternative is to write the output rows to a temporary table and sort that.)

The optimizer has a fourth option—performing a sort merge join—which does not require a temporary index. This occurs only when at least one filter of the join is an equality operator, and it has been determined that a temporary index would be more costly.

Selecting the Query Plan

With all this information available, the optimizer generates all possible query plans for joining tables in pairs. Then, if more than two tables are joined, the optimizer uses the best two-table plans to form all plans for joining two tables to a third, three to a fourth, and so on.

The optimizer adds any final work to the completed join plans. For example, if the query contains an ORDER BY or GROUP BY clause, and a plan does not produce rows in ordered sequence, the optimizer adds to the plan an estimate of the cost of sorting the output. Sorting is discussed in “Time Cost of a Sort” on page 13-15.

Finally, the optimizer selects a plan that appears to promise the least amount of work, and passes it to the main part of the database server to be executed.

Reading the Plan

The choice the optimizer makes does not have to remain a mystery; you can determine exactly what query plan it chooses. Execute the statement SET EXPLAIN ON before you execute a query. Beginning with the next query, the optimizer writes an explanation of its query plan to a particular file (the name of the file and its location depend on the operating system in use). A typical explanation is shown in Figure 13-1.

After repeating the query, the optimizer shows its estimate of the work to be done (102 in Figure 13-1) in arbitrary units. A single disk access is one unit, and other actions are scaled to that. This query plan was chosen over others because the estimated cost for its execution was the lowest among all the plans considered.

```

QUERY:
-----
SELECT C.customer_num, O.order_num, SUM (I.total_price)
  FROM customer C, orders O, items I
 WHERE C.customer_num = O.customer_num
       AND O.order_num = I.order_num
 GROUP BY C.customer_num, O.order_num;

Estimated Cost: 102
Estimated # of Rows Returned: 1
Temporary Files Required For: GROUP BY

1) pubs.o: SEQUENTIAL SCAN

2) pubs.c: INDEX PATH

   (1) Index Keys: customer_num (Key-Only)
   Lower Index Filter: pubs.c.customer_num = pubs.o.customer_num

3) pubs.i: INDEX PATH

   (1) Index Keys: order_num
   Lower Index Filter: pubs.i.order_num = pubs.o.order_num

```

Figure 13-1 Typical output produced by the optimizer with **SET EXPLAIN ON**

The optimizer also reveals its estimate of the number of rows that the query produces. In Figure 13-1, the optimizer incorrectly estimates 1. In this example, the estimate is incorrect because the optimizer has no way to estimate the number of groups the **GROUP BY** clause produces.

In the body of the explanation, the optimizer lists the order in which tables are accessed and the method, or access path, by which it reads each table. The following breakdown clarifies the plan:

1. The **orders** table is read first. Because no filter exists on the **orders** table, all of its rows must be read, that is, reading the table in physical order is the “least costly” approach.
2. For each row of **orders**, a search is made for matching rows in the **customer** table. The search uses the index on **customer_num**.

The notation *Key-Only* means that because only the **c.customer_num** column is used in the join and output, only the index is read; no row is read from the table.

3. For each row of **orders** that has a matching **customer_num**, a search is made in the **items** table using the index on **order_num**.

It is not always obvious why the optimizer makes its choices. By comparing the plans produced by several variations in the query, you can usually deduce some of its logic.

Time Costs of a Query

To execute a query, the database server spends most of its time in performing two types of operation: reading data from disk, and comparing column values. Of the two, reading data is, by far, the slower task. This section examines where the database server spends time. The next section explores the implications of making queries faster.

Activities in Memory

The database server can process only data in memory. It must read a row into memory before it can test it with a filter expression. It must read rows from both tables before it can test a join condition. The database server prepares an output row in memory by assembling the selected columns from other rows in memory.

Most of these activities execute very quickly. Depending on the computer, the database server can perform hundreds or even thousands of comparisons each second. As a result, the time spent on in-memory work is usually a small part of the whole execution time.

Two in-memory activities can take a significant amount of time. One is sorting, as described in “Time Cost of a Sort” on page 13-15. The other activity is processing comparisons using LIKE and MATCHES, when they begin with “wild card” characters.

Disk-Access Management

It takes much longer to read a row from disk than to examine a row in memory. The main goal of the optimizer is to reduce the amount of data that must be read from disk, but it can eliminate only the most obvious inefficiencies.

Disk Pages

The database server deals with disk storage in units called *pages*. A page is a block of fixed size. The same size is used for all databases managed by one database server. Indexes are stored in page-size units.

Time Cost of a Sort

A sort requires in-memory work as well as disk work.

The in-memory work is proportional to $c \cdot w \cdot n \cdot \log_2(n)$, where

c is the number of columns being ordered, and represents the costs of extracting column values from the row and concatenating them into a sort key.

w is proportional to the width of the combined sort key in bytes, and stands for the work of copying or comparing one sort key. A numeric value for w would depend strongly on the computer hardware in use.

$n \cdot \log_2(n)$ is the number of comparisons that are made while sorting a table of n rows.

The disk work is proportional to $2 \cdot n \cdot m$, where n is again the number of rows. The factor m represents the number of *levels of merge* the sort must use, a factor that depends on the number of sort keys that can be held in memory.

When all the keys can be held in memory, $m=1$ and the disk work is proportional to $2n$. In other words, the rows are read, sorted in memory, and written. (When building an index, only index pages are written.)

For tables from moderate to large sizes, rows are sorted in batches that fit in memory and then the batches are merged. When $m=2$, the rows are read, sorted, and written in batches. Then the batches are read again and merged and written again, resulting in disk work proportional to $4n$. For extremely large tables, the batches must be broken into batches. The disk work is then proportional to $6n$. In short, as table size increases, there are sudden, discontinuous changes in the amount of disk work in a sort, from $2n$ to $4n$ to $6n$. The table size at which these steps occur depends on many factors, only one of which (key size) is in your control.

The best way to reduce the cost of sorting is to find a way to sort fewer rows, because the factor n dominates both expressions. When that is not possible, look for ways to sort on fewer and narrower columns. Not only does this reduce the factors c and w it also defers the step to the next merge level.

Sorting involves writing information temporarily to disk. You can direct the disk writes to occur in a dbspace managed by **INFORMIX-OnLine Dynamic Server**, or an operating system file. Writing to a dbspace often is much faster than writing to an operating system file. You use the PSORT_DBTEMP environment variable to specify one or more directories where temporary information is stored in operating system files in a round robin manner. If PSORT_DBTEMP is not set, temporary information is written to the dbspaces specified by the DBSPACETEMP environment variable, in a round robin manner. If both PSORT_DBTEMP and DBSPACETEMP are not set, temporary information is written to the dbspaces specified by the DBSPACETEMP ONCONFIG parameter, in a round robin manner. If none of these environment variables or ONCONFIG parameter are set, temporary information is written to operating system files in /tmp. For additional information about environment variables, see Chapter 4 of *Informix Guide to SQL: Reference*. For additional information about the ONCONFIG file, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

The size of a page depends on the database server. With **INFORMIX-OnLine Dynamic Server**, the page size is set when **OnLine** is initialized. It is usually 2 kilobytes (2,048 bytes), but you can ask the person who installed **OnLine** to tell you what was chosen. Other Informix database servers use the file storage of the host operating system, so their page size is the block size used by the host operating system. One kilobyte (1,024 bytes) is typical.

It is possible to define tables so wide that one row fills a page (some database servers permit a row to exceed the size of a page). However, a typical disk page contains from 5 to 50 rows. An index entry consists of a key value and a 4-byte pointer, so an index page typically contains from 50 to 500 entries.

Page Buffers

The database server has a set of memory spaces in which it keeps copies of the disk pages it read most recently. It does this in case these pages are needed again. If they are, the database server does not have to read them from disk.

As with the size of a disk page, the number of these page buffers depends on the database server and the host operating system.

The Cost of Reading a Row

When the database server needs to examine a row that is not already in memory, it must read it from disk. It does not read only one row; it reads the pages that contain the row. (When a row is larger than a page, it reads as many whole pages as necessary.) The cost of reading one page is the basic unit of work that the optimizer uses for its calculations.

The actual cost of reading a page is variable and hard to predict. It is a combination of the following factors:

Buffering	The needed page might be in a page buffer already, in which case the cost of access is near zero.
Contention	If more than one application is contending for the use of the disk hardware, the database server request can be delayed.
Seek time	The slowest thing a disk does is to <i>seek</i> ; that is, to move the access arm to the track holding the data. Seek time depends on the speed of the disk and the location of the disk arm when the operation starts. Seek time varies from zero to a large fraction of a second.
Latency	The transfer cannot start until the beginning of the page rotates under the access arm. This <i>latency</i> , or rotational delay, depends on the speed of the disk and on the position of the disk when the operation starts. Latency can vary from zero to a few milliseconds.

The time cost of reading a page can vary from microseconds (for a page in a buffer), to a few milliseconds (when contention is zero and the disk arm is already in position), to hundreds of milliseconds.

The Cost of Sequential Access

Disk costs are lowest when the database server reads the rows of a table in physical order. When the first row on that page is requested, its disk page is read. Requests for subsequent rows are satisfied from the buffer until the entire page is used. As a result, each page is read only once.

Provided that the database server is the only program using the disk, the seek-time cost is also minimized. The disk pages of consecutive rows are usually in consecutive locations on the disk, so the access arm moves very little from one page to the next. Even when that is not the case, groups of consecutive pages are usually located together so that only a few long seeks are needed. For example, **INFORMIX-OnLine Dynamic Server** allocates disk space to a table in multipage *extents*. Extents may be separated on the disk, but within an extent the pages are close together.

Even latency costs may be lowest when pages are read sequentially. This depends on the hardware and, when operating system files are used, on the methods of the operating system. Disks are usually set up so that when pages are read sequentially, latency is minimized. On the other hand, it is possible to set up a disk so that a full rotation occurs between each sequential page, drastically slowing sequential access.

The Cost of Nonsequential Access

Disk costs are higher when the rows of a table are called for in a sequence that is unrelated to physical order. Practical tables are normally much larger than the database server page buffers, so only a small part of the table pages can be held in memory. When a table is read in nonsequential order, only a few rows are found in buffered pages. Usually one disk page is read for every row that is requested.

Because the pages are not taken sequentially from the disk, usually both a seek delay and a rotational delay occur before each page can be read. In short, when a table is read nonsequentially, the disk-access time is much higher than when it is read sequentially.

The Cost of Rowid Access

The simplest form of nonsequential access is to select a row based on its *rowid* value. (The use of rowid in a SELECT statement is discussed in Chapter 3, “Composing Advanced SELECT Statements.”) A rowid value specifies the physical location of the row and its page. The database server simply reads the page, incurring the costs already noted.

The Cost of Indexed Access

An additional cost is associated with finding a row through an index: The index itself is stored on disk, and its pages must be read into memory.

The database server uses indexes in two ways. One way is to look up a row given a key value. This is the kind of look up used when you join two tables, as shown in the following statement:

```
SELECT company, order_num
FROM customer, orders
WHERE customer.customer_num = orders.customer_num
```

One table, probably **customer**, is read sequentially. Its value of **customer_num** is used to search the index on the **customer_num** column of **orders**. When a match is found, that row of **orders** is read.

An index look up works down from the root page to a leaf page. (See “The Structure of an Index” on page 13-20.) The root page, because it is used so often, is almost always found in a page buffer. The odds of finding a leaf page in a buffer depend on the size of the index; the odds become poorer as the size of the table increases.

If a table is very large, so that the number of index leaf pages is much larger than the buffer space, almost every search causes a leaf page to be read in addition to the page containing the row. The number of pages read is approximately $r * (\text{index} - \text{height} - 1)$, where r is the number of rows to be looked up. Although it is costly, this kind of access is still a bargain, because the alternative is to perform each look up by reading the entire **orders** table, with disk costs proportional to r^2 .

The other way the database server uses an index is to read it sequentially. It does this to fetch the rows of a table in a specific order other than the physical order. For this kind of access, the database server reads the leaf pages in sequence, treating them as a list of rows in key sequence. Because the index pages are read only once, the total of disk operations is proportional to $r + i$; i is the number of leaf pages in the index.

The Cost of Small Tables

One conclusion you can draw from the preceding paragraphs is that small tables are never slow. A table is small if it occupies so few pages that it can be retained entirely in the page buffers. Any table that fits in four or fewer pages is certainly small in this sense.

In the **stores6** database, the **state** table that relates abbreviations to names of states has a total size less than 1,000 bytes; it fits in, at most, two pages. It can be included in any query at little cost. No matter how it is used—even if it is read sequentially many times—it costs two disk accesses at most.

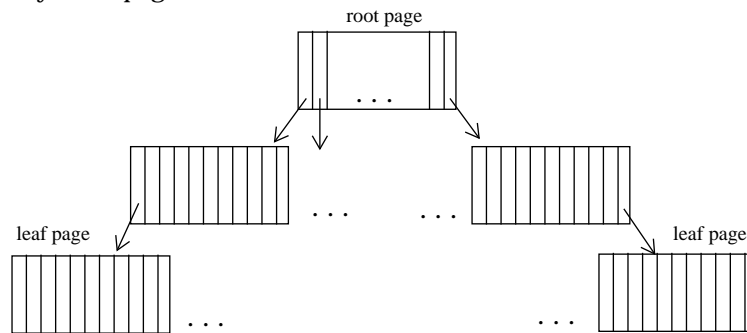
The Cost of NLS Functionality

The sorting and indexing of data containing foreign characters can cause significant performance degradation for certain data sets. If you do not need the functionality of NLS for all your character data, Informix recommends that you use the CHAR data type instead of NCHAR (or VARCHAR instead of NVARCHAR) for character columns whenever possible. Because CHAR (and VARCHAR) data require simple value-based comparison, sorting, and indexing, the cost of CHAR (or VARCHAR) columns is thus relatively less expensive than NCHAR (or NVARCHAR) columns.

The Structure of an Index

An index is arranged as a hierarchy of pages (technically, a *B+ tree*), as depicted in the figure below. At the bottom level, a sequence of *leaf pages* contains the index key values of the table rows, each with a pointer to the location of the row containing that value. The keys in the leaf pages are in key sequence.

At each higher level, a page lists the highest key value to be found in each page at the lower next level. The topmost level is a single *root page* from which any other page of the index can be found.



The size of one entry is the width of the indexed columns in bytes, plus 4. If you know that and the page size, you can estimate the number of entries per page. This estimate is approximate because first, some data compression is used in storing keys and second, not all pages are full.

The number of levels depends on the number of rows to be indexed and the number of entries per page. Suppose a page holds 100 keys. Then a table of 100 rows has a single index page, the root page. A table with between 101 and 10,000 rows has a two-level index consisting of a root page and from 2 to 100 leaf pages. A table with as many as 1,000,000 rows has only a 3-level index (so the preceding index must belong to a table of considerable size).

The Cost of Network Access

Whenever data is moved over a network, additional delays are imposed. Networks are used in the following contexts:

- The application sends a query across the network to a database server in another machine. The database server performs the query using its locally

attached disk. The output rows are returned over the network to the application.

- Using the distributed data capability of **INFORMIX-OnLine Dynamic Server**, a database server in one machine can read and update rows from tables in databases located on other machines.

The data sent over a network consists of command messages and buffer-sized blocks of row data. Although many differences of detail exist between the two contexts, they can be treated identically under a simple model in which one machine, the *sender*, sends a request to another, the *responder*, which responds with a block of data from a table.

Whenever data is exchanged over a network, delays are inevitable in the following situations:

- If the network is busy, the sender must wait its turn to transmit. Such delays are typically less than a millisecond. But in a heavily loaded network, they can increase exponentially to tenths of seconds and more.
- The responder may be handling requests from more than one sender, so when the request arrives, it may be queued for a time that can range from milliseconds to seconds.
- When the responder acts on the request, it incurs the time costs of disk access and in-memory operations as described in the preceding topics.
- Transmission of the response is again subject to network delays.

The important point about network access is its extreme variability. In the best case, when neither the network nor the responder is busy, transmission and queueing delays are insignificant and the responder sends a row almost as quickly as a local database server could do it. Furthermore, when the sender asks for a second row, the page is likely still to be in the responder page buffers.

Unfortunately, as network load increases, all these factors tend to worsen at the same rate. Transmission delays rise in both directions. The queue at the responder gets longer. And the odds of a page remaining in the responder buffer become worse. Thus, network access costs can change quite suddenly from very low to extremely high.

The optimizer that **INFORMIX-OnLine Dynamic Server** uses assumes that access to a row over the network takes longer than access to a row in a local database. This estimate includes the cost of retrieving the row from disk and transmitting it across the network. The optimizer has no way to account for varying network loads, so it is too pessimistic at times and too optimistic at others.

The Importance of Table Order

The order in which tables are examined has an enormous effect on the speed of a join operation. This concept can be clarified with some examples that show how the database server works.

A Join Without Filters

The following SELECT statement calls for a three-way join:

```
SELECT C.customer_num, O.order_num, SUM (I.total_price)
FROM customer C, orders O, items I
WHERE C.customer_num = O.customer_num
      AND O.order_num = I.order_num
GROUP BY C.customer_num, O.order_num
```

For the moment, imagine that indexes were never invented. Without indexes, the database server has no choice but to perform this operation using a simple nested loop. One of two practical *query plans* is displayed in Figure 13-2, which is expressed in a programming pseudocode. A query plan states the order in which the database server examines tables and the methods by which it accesses the tables.

```
for each row in the customer table do:
  read the row into C
  for each row in the orders table do:
    read the row into O
    if O.customer_num = C.customer_num then
      let Sum = 0
      for each row in the items table do:
        read the row into I
        if I.order_num = O.order_num then
          let Sum = Sum + I.total_price
        end if
      end for
      prepare an output row from C,I,and Sum
    end if
  end for
end for
```

Figure 13-2 *A query plan in pseudocode*

This procedure reads the following rows:

- All rows of the **customer** table once
- All rows of the **orders** table once for each row of the **customer** table
- All rows of the **items** table once for each row of **orders** that appears in the output (every row of **orders** should appear once)

This is not the only possible query plan; another merely reverses the roles of **customer** and **orders**: for each row of **orders**, it reads all rows of **customer** looking for a matching **customer_num**. It reads the same number of rows, but in a different sequence.

A Join with Column Filters

In the preceding example, no difference exists in the amount of work that the two possible query plans need to do. The presence of a *column filter* changes things. A column filter is a WHERE expression that reduces the number of rows that a table contributes to a join. The following example shows the preceding query with a filter added:

```
SELECT C.customer_num, O.order_num, SUM (I.total_price)
FROM customer C, orders O, items I
WHERE C.customer_num = O.customer_num
      AND O.order_num = I.order_num
      AND O.paid_date IS NULL
GROUP BY C.customer_num, O.order_num
```

The expression `O.paid_date IS NULL` *filters out* some rows, reducing the number of rows that are used from the **orders** table. As before, two query plans are possible. The plan that starts by reading from **orders** is displayed in pseudocode in Figure 13-3.

```
for each row in the orders table do:
  read the row into O
  if O.paid_date is null then
    for each row in the customer table do:
      read the row into C
      if O.customer_num = C.customer_num then
        let Sum = 0
        for each row in the items table do:
          read the row into I
          if I.order_num = O.order_num then
            let Sum = Sum + I.total_price
          end if
        end for
        prepare an output row from C,I,and Sum
      end if
    end for
  end if
end for
```

Figure 13-3 *One of two query plans in pseudocode*

Let *pdnull* represent for the number of rows in **orders** that pass the filter. It is the number that the following query returns:

```
SELECT COUNT(*) FROM orders WHERE paid_date IS NULL
```

Assume that just one customer exists for every order, as there should be. Then we can say that the plan in Figure 13-3 reads the following rows:

- All rows of the **orders** table once
- All rows of the **customer** table, *pdnull* times
- All rows of the **items** table, *pdnull* times

An alternative plan is shown in Figure 13-4; it reads from **customer** first.

```
for each row in the customer table do:
  read the row into C
  for each row in the orders table do:
    read the row into O
    if O.paid_date is null and
       O.customer_num = C.customer_num then
      let Sum = 0
      for each row in the items table do:
        read the row into I
        if I.order_num = O.order_num then
          let Sum = Sum + I.total_price
        end if
      end for
      prepare an output row from C,I,and Sum
    end if
  end for
end for
```

Figure 13-4 *The alternative query plan in pseudocode*

Because the filter is not applied in the first step, shown in Figure 13-4, this plan reads the following rows:

- All rows of the **customer** table once
- All rows of the **orders** table, once for every row of **customer**
- All rows of the **items** table, *pnull* times

The query plans in Figure 13-3 and Figure 13-4 produce the same output, although in different sequences. They differ in that one reads a table *pnull* times and the other reads a table `SELECT COUNT(*) FROM customer` times. The choice the optimizer makes between them could make a difference of thousands of disk accesses in a real application.

Using Indexes

The preceding examples do not use indexes or constraints, which is unrealistic. Almost all tables have one or more indexes or constraints, and their presence makes a difference in the query plan. Figure 13-5 shows the outline of a query plan for the previous query as it might be constructed using indexes. (Also see “The Structure of an Index” on page 13-20.)

```
for each row in the customer table do:
  read the row into C
  look up C.customer_num in index on orders.customer_num
  for each matching row in the orders table do:
    read the row into O
    if O.paid_date is null then
      let Sum = 0
      look up O.order_num in index on items.order_num
      for each matching row in the items table do:
        read the row into I
        let Sum = Sum + I.total_price
      end for
      prepare an output row from C,I,and Sum
    end if
  end for
end for
```

Figure 13-5 *A query plan using indexes in pseudocode*

The keys in an index are sorted so that when the first matching key is found, any other rows with identical keys can be read without further searching. This query plan reads only the following rows:

- All rows of the **customer** table once
- All rows of the **orders** table once
- Those rows of the **items** table that match to *pdnull* rows from **orders**

This is a huge reduction in effort compared with the plans without indexes. (An inverse plan, reading **orders** first and looking up rows in **customer** by its index, is equally effective.)

However, any plan that uses an index must read index data from disk as well as read row data. It is difficult to predict the number of index pages that are read, because some number of recently used index pages are retained in memory and are found there when they are needed. (See “Disk-Access Management” on page 13-14.)

The physical order of a table also affects the cost of index use. The physical order of the **customer** table is customer-number order if a clustered index is on that column. It is approximately that order if customer numbers are SERIAL values generated by the database server.

However, if the physical order is random with respect to customer numbers, each search of the index on **orders** might lead to a different page of index keys, so that an index page is read for almost every row.

Note that when NLS is enabled, indexes built on NCHAR or NVARCHAR columns are sorted using a country-specific comparison value. For example, the Spanish double-l character (ll) may be treated as a single unique character instead of a pair of l's.

The Sort-Merge Join Technique

The sort-merge table join provides a cost-effective alternative to constructing a temporary index for a nested-loop join. It is considered by the optimizer only when the join is based on an equality filter and no index exists that can accomplish the join.

The sort-merge join does not change the fundamental strategy of the optimizer. Rather, it provides a richer set of alternatives to nested-loop joins using temporary indexes and alters the way the ORDER BY and GROUP BY paths are analyzed. The path chosen by the optimizer is displayed in the output when the SET EXPLAIN ON statement is issued, as described in Chapter 1 of the *Informix Guide to SQL: Syntax* and in “Reading the Plan” on page 13-12 of this manual.

Making Queries Faster

In general, you can speed up a query by changing it so that it performs the following actions:

- Reads fewer rows
- Avoids a sort, sorts fewer rows, or sorts on a simpler key
- Reads rows sequentially instead of nonsequentially

The way to achieve these ends is not always obvious. The specific methods depend on the details of the application and the database design. The following paragraphs suggest a general approach, followed by some techniques that apply in limited circumstances.

Preparing a Test Environment

First, select a single query that is too slow. Then set up an environment in which you can take predictable, repeatable timings of that query. Without this environment, you can never be sure whether a change helps.

If you are using a multiuser system or a network, so that system load varies widely from hour to hour, you may need to perform your experiments at the same time each day to obtain repeatable results. Preferably, you should execute tests when the system load is consistently light so that you are truly measuring only your query.

If the query is embedded in a complicated program, consider extracting the SELECT statement and executing it interactively or embedding it in a simpler program.

If the real query takes many minutes or hours to complete, it may be a good idea to prepare a scaled-down database in which you can run tests more quickly. This can be helpful, but you must be aware of the following potential problems:

- The optimizer can make different choices in a small database than in a large one, even when the relative sizes of tables are the same. Verify that the query plan is the same in the real and the model databases.
- Execution time is rarely a linear function of table size. Sorting time, for example, increases faster than table size, as does the cost of indexed access when an index goes from two to three levels. What appears to be a big improvement in the scaled-down environment can be insignificant when applied to the full database.

Therefore, any conclusion you reach as a result of tests in the model database must be tentative until verified in the large one.

Studying the Data Model

Study the definitions of all the tables, views, and indexes used in the database. You can examine these details interactively using the Table option of **DB-Access** or **INFORMIX-SQL**. Pay particular attention to the indexes that exist, to the data types of columns used in join conditions and for ordering, and to the presence of views. This chapter is written assuming that you cannot change the data model. Nevertheless, the better you understand it, the better you can understand the query plans chosen by the optimizer.

For information on data types and views, see Chapter 9, “Implementing Your Data Model.”

Studying the Query Plan

Use the output from SET EXPLAIN ON to determine the query plan being used. The following list contains some things to look for:

Indexes	Examine the output to see if indexes are used and if so, how they are used.
Filters	Examine your filters to make sure that they are selective enough.
Query	Examine your query again to make sure that the access plan is optimal.

Examining the Indexes

You should study the indexes or the lack of indexes on the columns of your query. Consider using some of the methods that are described in the following sections to improve your query.

Replacing Autoindexes with Indexes

If the query plan includes an *autoindex* path to a large table, take it as a recommendation from the optimizer that an index should be on that column. It is reasonable to let the database server build and discard an index if you perform the query only occasionally, but if the query is done even daily, you save time by creating a permanent index.

Creating Data Distributions on Filtered Columns

Executing the UPDATE STATISTICS statement in MEDIUM or HIGH mode causes information about the distribution of data values for each column specified to be placed in a system catalog table called **sysdistrib**. This information is used at optimization time to estimate the number of rows that match a given query, and from that, to estimate the cost of retrieving those values. From these cost estimates, the optimizer determines the best path of execution. Data distributions supercede the **colmin** and **colmax** columns in the **syscolumns** system catalog table in estimating the number of rows returned by each part of a WHERE clause.

When using data-distribution statistics for the first time, try updating statistics in MEDIUM mode on all tables followed by updating statistics in HIGH mode on columns that head indexes. This will produce single-table query estimates that, on the average, have an error less than 2.5 percent of the total number of rows in the table. (For columns with HIGH mode distributions, the default resolution is 1.0 percent.)

Unless column values change by a considerable margin, it is unnecessary to regenerate data distributions. You can verify the accuracy of the distribution by comparing **dbschema** (-hd option) output with appropriately constructed SELECT statements.

For additional information about data distributions and the UPDATE STATISTICS statement, see Chapter 1 of the *Informix Guide to SQL: Syntax*.

Filter Selectivity Assignments

The following table lists some of the selectivities that the optimizer assigns to filters of different types. It is not an exhaustive list, and other expression forms may be added in the future.

Expression Form	Selectivity (F)
<i>indexed-col</i> = <i>literal-value</i>	$F = 1 / (\text{number of distinct keys in index})$
<i>indexed-col</i> = <i>host-variable</i>	
<i>indexed-col</i> IS NULL	$F = 1 / (\text{number of distinct keys in the larger index})$
<i>tab1.indexed-col</i> = <i>tab2.indexed-col</i>	$F = (2nd\text{-}max - \text{literal-value}) / (2nd\text{-}max - 2nd\text{-}min)$
<i>indexed-col</i> > <i>literal-value</i>	$F = (\text{literal-value} - 2nd\text{-}min) / (2nd\text{-}max - 2nd\text{-}min)$
<i>indexed-col</i> < <i>literal-value</i>	
<i>any-col</i> IS NULL	$F = 1/10$
<i>any-col</i> = <i>any-expression</i>	
<i>any-col</i> > <i>any-expression</i>	$F = 1/3$
<i>any-col</i> < <i>any-expression</i>	
<i>any-col</i> MATCHES <i>any-expression</i>	$F = 1/5$
<i>any-col</i> LIKE <i>any-expression</i>	
EXISTS <i>subquery</i>	$F = 1$ if <i>subquery</i> estimated to return >0 rows, else 0
NOT <i>expression</i>	$F = 1 - F(\text{expression})$
<i>expr1</i> AND <i>expr2</i>	$F = F(\text{expr1}) \times F(\text{expr2})$
<i>expr1</i> OR <i>expr2</i>	$F = F(\text{expr1}) + F(\text{expr2}) - F(\text{expr1}) \times F(\text{expr2})$
<i>any-col</i> IN <i>list</i>	treated as <i>anycol</i> = <i>item</i> ₁ OR...OR <i>anycol</i> = <i>item</i> _n
<i>any-col</i> relop ANY <i>subquery</i>	treated as <i>any-col</i> relop <i>value</i> ₁ OR...OR <i>anycol</i> relop <i>value</i> _n for estimated size of subquery <i>n</i>
Key:	
<i>indexed-col</i> : first or only column in an index (OnLine only)	
<i>2nd-max</i> , <i>2nd-min</i> : second-largest and -smallest key values in indexed column (OnLine only)	
<i>any-col</i> : any column not covered by a preceding formula	

Selectivities are more accurate with distributions than through current index information due to finer granularity statistics. However, if data distributions are not used, filter selectivities are used as shown in the following chart.

Using Composite Indexes

The optimizer can use a composite index (one that covers more than one column) in several ways. In addition to its usual function of ensuring that the values in columns *abc* are unique, an index on the columns *a*, *b*, and *c* (in that order) can be used in the following ways:

- To evaluate filter expressions on column *a*
- To join column *a* to another table
- To implement ORDER BY or GROUP BY on columns *a*, *ab*, or *abc* (but not on *b*, *c*, *ac*, or *bc*)

When you create a composite index, place the columns in the index in the order from most frequently used to least frequently used.

If your application is performing several long queries, each of which involves a sort on the same columns, you might save time by creating a composite index on those columns. In effect, you perform the sort once and save its output for use in every query.

Using *oncheck* on Suspect Indexes

With some database servers, it is possible for an index to become ineffective because it has been internally corrupted. If a query that uses an index has slowed down inexplicably, use the **oncheck** utility to check the integrity of the index and to repair it if necessary. (The **secheck** utility does the same job for INFORMIX-SE.)

Dropping and Rebuilding Indexes After Updates

After extensive amounts of updating (after the replacement of a fourth or more of the rows of a table), the structure of an index can become inefficient. If an index seems to be less effective than it should be, yet **oncheck** reports no errors, try dropping the index and re-creating it.

Refining the Column Filters

The most efficient queries access only a small body of information. You control the amount of information returned in the SELECT statement with WHERE clauses, which are commonly called filters. These sections suggest the following ways to improve the WHERE clause of your query:

- Avoiding correlated subqueries
- Avoiding difficult regular expressions

- Avoiding noninitial substrings

Avoiding Correlated Subqueries

A correlated subquery is one in which a column label appears in both the select list of the main query and the WHERE clause of the subquery. Because the result of the subquery might be different for each row that the database server examines, the subquery executes anew for every row if the current correlation values are different from the previous ones. The optimizer tries to use an index on the correlation values to cluster identical values together. This procedure can be extremely time consuming. Although some queries cannot be stated in SQL without the use of a correlated subquery, many can be rewritten as joins that will execute more rapidly.

When you see a subquery in a time-consuming SELECT statement, look to see if it is correlated. (An uncorrelated subquery, one in which no row values from the main query are tested within the subquery, is executed only once.) If so, try to rewrite the query to avoid it. If you cannot, look for ways to reduce the number of rows that are examined; for instance, try adding other filter expressions to the WHERE clause, or try selecting a subset of rows into a temporary table and searching only them.

Avoiding Difficult Regular Expressions

The MATCHES and LIKE keywords support *wildcard* matches—technically known as *regular expressions*. Some regular expressions are more difficult than others for the database server to process. A wildcard in the initial position, as in the following example (find customers whose first names do not end in y), forces the database server to examine every value in the column, as shown in the following example:

```
SELECT * FROM customer WHERE fname NOT LIKE '%y'
```

An index cannot be used with such a filter so the table in this example must be accessed sequentially.

If a difficult test for a regular expression is essential, avoid combining it with a join. If necessary, process the single table, applying the test for a regular expression to select the desired rows. Save the result in a temporary table, and join that table to the others.

Regular-expression tests with wildcards in the middle or at the end of the operand do not prevent the use of an index when one exists.

Avoiding Noninitial Substrings

A filter based on a noninitial substring of a column also requires every value in the column to be tested, as shown in the following example:

```
SELECT * FROM customer
WHERE zipcode[4,5] > '50'
```

An index cannot be used to evaluate such a filter.

The optimizer uses an index to process a filter that tests an initial substring of an indexed column. However, the presence of the substring test can interfere with the use of a composite index to test both the substring column and another column.

Rethinking the Query

After you understand what the query is doing, look for ways to obtain the same output with less effort. The following suggestions can help you rewrite your query more efficiently:

- Rewriting joins through views
- Avoiding or simplifying sorts
- Eliminating sequential access to large tables
- Using unions to avoid sequential access

Rewriting Joins Through Views

You might find that the query joins a table to a view that is itself a join. If you rewrite the query to join directly to fewer tables, you might produce a simpler query plan.

Avoiding or Simplifying Sorts

A sort is not necessarily negative. The sort algorithm of the database server is highly tuned and extremely efficient. It is certainly as fast as any external sort program you might apply to the same data. As long as the sort is performed only occasionally or to a relatively small number of output rows, no need exists to avoid it.

However, you should avoid or simplify repeated sorts of large tables. The optimizer avoids a sort step whenever it can produce the output in its proper order automatically by using an index. The following factors prevent the optimizer from using an index:

- One or more of the ordered columns is not included in the index.
- The columns are named in a different sequence in the index and the ORDER BY or GROUP BY clause.
- The ordered columns are taken from different tables.

Another way to avoid sorts is discussed in “Using a Temporary Table to Speed Queries” on page 13-36.

If a sort is necessary, look for ways to simplify it. As discussed in “Time Cost of a Sort” on page 13-15, the sort is quicker if you can sort on fewer or narrower columns.

Eliminating Sequential Access to Large Tables

Sequential access to a table other than the very first one in the plan is ominous; it threatens to read every row of the table once for every row selected from the preceding tables. You should be able to judge how many times that is: perhaps a few, but perhaps hundreds or even thousands.

If the table is small, it can do no harm to read it repeatedly; the table resides completely in memory. Sequential search of an in-memory table can be faster than searching the same table through an index, especially if also having its index pages in memory pushes other useful pages out of the buffers.

When the table is larger than a few pages, however, repeated sequential access is deadly to performance. One way to prevent this is to provide an index to the column being used to join the table.

The subject of indexing is discussed in the context of database design in “Managing Indexes” on page 10-19. However, any user with Resource privileges can build additional indexes. Use the CREATE INDEX statement to make an index.

An index consumes disk space proportional to the width of the key values and the number of rows. (See “The Structure of an Index” on page 13-20.) Also, the database server must update the index whenever rows are inserted, deleted, or updated; this slows these operations. If necessary, you can use the DROP INDEX statement to release the index after a series of queries, thus freeing space and making table updates easier.

Using Unions to Avoid Sequential Access

Certain forms of WHERE clauses force the optimizer to use sequential access even when indexes exist on all the tested columns. The following query forces sequential access to the **orders** table:

```
SELECT * FROM orders
  WHERE (customer_num = 104 AND order_num > 1001)
        OR order_num = 1008
```

The key element is that two (or more) separate sets of rows are retrieved. The sets are defined by relational expressions that are connected by OR. In the following example, one set is selected by this test:

```
(customer_num = 104 AND order_num > 1001)
```

The other set is selected by the test shown in the following example:

```
order_num = 1008
```

The optimizer uses a sequential access path even though indexes exist on the **customer_num** and **order_num** columns.

You can speed queries of this form by converting them into UNION queries. Write a separate SELECT statement for each set of rows and connect them with the UNION keyword. The following example shows a rewritten version of the preceding example:

```
SELECT * FROM orders
  WHERE (customer_num = 104 AND order_num > 1001)

UNION

SELECT * FROM orders
  WHERE order_num = 1008
```

The optimizer uses an index path for each query.

You can also rewrite the WHERE clause as shown in the following example:

```
WHERE (customer_num = 104 OR order_num = 1008) AND
      (order_num > 1001 OR order_num = 1008)
```

Reviewing the Optimization Level

You normally obtain optimum overall performance with the default optimization level, HIGH. The time it takes to optimize the statement is usually unimportant. However, if experimentation with your application reveals that your query is still taking too long, you may want to set your optimization level to LOW, and then check the SET EXPLAIN output to see if the optimizer chose the same query plan as before.

You can specify a HIGH or LOW level of database server optimization with the SET OPTIMIZATION statement. This statement is described in detail in Chapter 1 of the *Informix Guide to SQL: Syntax*.

Using a Temporary Table to Speed Queries

Building a temporary, ordered subset of a table can speed up a query. It can help to avoid multiple-sort operations and can simplify the work of the optimizer in other ways.

Using a Temporary Table to Avoid Multiple Sorts

For example, suppose your application produces a series of reports on customers who have outstanding balances, one report for each major postal area, ordered by customer name. In other words, a series of queries occur each of the following form (using hypothetical table and column names):

```
SELECT cust.name, rcvbles.balance, ...other columns...
FROM cust, rcvbles
WHERE cust.customer_id = rcvbles.customer_id
      AND rcvbles.balance > 0
      AND cust.postcode LIKE '98_ _ _'
ORDER BY cust.name
```

This query reads the entire **cust** table. For every row with the right postcode, the database server searches the index on **rcvbles.customer_id** and performs a nonsequential disk access for every match. The rows are written to a temporary file and sorted.

This procedure is acceptable if the query is done only once, but this example includes a series of queries, each incurring the same amount of work.

An alternative is to select all customers with outstanding balances into a temporary table, ordered by customer name, as shown in the following example:

```
SELECT cust.name, rcvbles.balance, ...other columns...
FROM cust, rcvbles
WHERE cust.customer_id = rcvbles.customer_id
      AND cvbbs.balance > 0
ORDER BY cust.name
INTO TEMP cust_with_balance
```

Now you can direct queries against the temporary table in this form, as shown in the following example:

```
SELECT *
FROM cust_with_balance
WHERE postcode LIKE '98_ _ _'
```

Each query reads the temporary table sequentially, but the table has fewer rows than the primary table. No nonsequential disk accesses are performed. No sort is required because the physical order of the table is the desired order. The total effort should be considerably less than before.

One possible disadvantage exists: Any changes made to the primary table after the temporary table has been created are not reflected in the output. This is not a problem for some applications, but it may be for some.

Substituting Sorting for Nonsequential Access

Nonsequential disk access is the slowest kind. SQL hides this fact and makes it easy to write queries that access huge numbers of pages nonsequentially. Sometimes you can improve a query by substituting the sorting ability of the database server for nonsequential access. The following example demonstrates this and also demonstrates how to make numerical estimates of the cost of a query plan.

Imagine a large manufacturing firm whose database includes three tables shown in schematic form in Figure 13-6. (Table diagrams in this form are described in Chapter 8, “Building Your Data Model.” Not all the columns are shown.)

The first table, **part**, contains the parts used in the products of the company.

The second, **vendor**, contains data about the vendors who provide the parts. The third, **parven**, records which parts are available from which vendors and at what price.

part		vendor	
part_num	part_desc	vendor_num	vendor_name
pk		pk	
100,000	spiral spanner	9,100,000	Wrenchers SA
999,999	spotted paint	9,999,999	Spottiswode

parven		
part_num	vendor_num	price
pk, fk	pk, fk	
100,000	9,100,000	\$14.98
999,999	9,999,999	\$0.066

Figure 13-6 Three tables from a manufacturing database

The following query is run regularly against these tables to produce a report of all available prices:

```
SELECT part_desc, vendor_name, price
FROM part, vendor, parven
WHERE part.part_num = parven.part_num
      AND parven.vendor_num = vendor.vendor_num
ORDER BY part.part_num
```

Although it appears to be a relatively simple three-table join, the query takes too much time. As part of your investigation, prepare a table showing the approximate sizes of the tables and their indexes in pages. The table appears in Figure 13-7. It is based on a disk page size of 4,096 bytes. Only approximate numbers are used. The actual number of rows changes often, and only estimated calculations are made in any case.

On further investigation, you learn that the index on **part_num** is clustered, so that the **part** table is in physical order by **part_num**. Similarly, the **vendor** table is in physical order by **vendor_num**. The **parven** table is not in any particular order. The sizes of these tables indicate that the odds of a successful nonsequential access from a buffered page are very poor.

The best query plan for this query (not necessarily the one the optimizer chooses) is to read the **part** table sequentially first, then use the value of **part_num** to access the matching rows of **parven** (about 1.5 of them per part), and then use the value of **parven.vendor_num** to access **vendor** through its index.

Tables	Row Size	Row Count	Rows/Page	Data Pages
part	150	10,000	25	400
vendor	150	1,000	25	40
parven	13	15,000	300	50

Indexes	Key Size	Keys/Page	Leaf Pages
part_num	4	500	20
vendor_num	4	500	2
parven (composite)	8	250	60

Figure 13-7 Documenting the sizes of tables and indexes in pages

The resulting number of disk accesses is estimated as shown in the following list:

- 400 pages read sequentially from **part**
- 10,000 nonsequential accesses to the **parven** table, 2 pages each (one index leaf page, one data page), or 20,000 disk pages
- 15,000 nonsequential accesses to the **vendor** table, or 30,000 disk pages

Even a simple join on well-indexed tables can require 50,400 disk reads. However, this can be improved by breaking the query into three steps using temporary tables as shown in Figure 13-8. (The following solution comes from W.H. Inmon; it is adapted from an example in his book *Optimizing Performance in DB2 Software*, Prentice-Hall, 1988.)

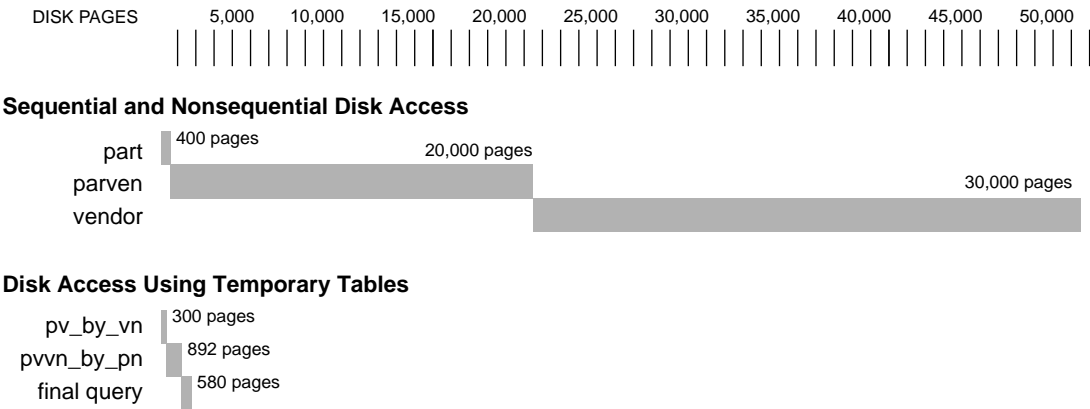


Figure 13-8 Breaking a query into three steps using temporary tables

The first step is to get the data of the **parven** table in **vendor_num** order, as shown in the following example:

```
SELECT part_num, vendor_num, price
FROM parven
ORDER BY vendor_num
INTO TEMP pv_by_vn
```

This statement reads **parven** sequentially (50 pages), writes a temporary table (50 pages), and sorts it. The cost of the sort, assuming one merge level, is about 200 pages, for a total of 300.

Join this temporary table to **vendor** and put the result in another temporary table, ordered by **part_num**, as shown in the following example:

```
SELECT pv_by_vn.*, vendor.vendor_name
FROM pv_by_vn, vendor
WHERE pv_by_vn.vendor_num = vendor.vendor_num
ORDER BY pv_by_vn.part_num
INTO TEMP pvn_by_pn;
DROP TABLE pv_by_vn
```


This query reads **pv_by_vn** sequentially (50 pages). It accesses **vendor** by way of its index 15,000 times, but as the keys are presented in **vendor_num** sequence, the effect is to read **vendor** sequentially through its index (42 pages).

If the **vendor_name** field is 32 bytes long, the output table has about 95 rows per page and occupies about 160 pages. These pages are written and then sorted, causing $5 \times 160 = 800$ page reads and writes. Thus, this query reads or writes 892 pages in all. Now join its output to **part** for the final result, which is shown in the following example:

```
SELECT pvvn_by_pn.*, part.part_desc
  FROM pvvn_by_pn, part
 WHERE pvvn_by_pn.part_num = part.part_num;
DROP TABLE pvvn_by_pn
```

This query reads **pvvn_by_pn** sequentially (160 pages). It accesses **part** by way of its index 15,000 times, but again the keys are presented in **part_num** sequence so the result is to read **part** sequentially (400 pages) through its index (20 pages). The output is produced in sorted order.

By splitting the query into three steps and by substituting sorts for indexed access, a query that read 50,400 pages is converted into one that reads and writes approximately 1,772 pages, a 30-to-1 ratio.

Note: Informix database servers earlier than version 4.1 do not permit you to use the *ORDER BY* and *INTO TEMP* clauses in the same query. With these database servers, you can achieve the preceding solution by selecting *INTO TEMP* without ordering, then applying the desired ordering through a clustered index. The improvement is, at most, 15:1 rather than of 30:1.

Summary

Poor performance can come from several sources, not solely the SQL operations in a program. Before you focus your attention on the code, take the following steps:

- Examine the application in its context of machines, people, procedures, and organization.
- Understand exactly what the application does, for whom, and why.
- Look for nontechnical solutions in the surrounding context.

- Develop a means of taking repeatable, quantitative measurements of the performance of the application.
- Isolate the time-consuming parts as narrowly as possible.

The performance of an individual query is determined by the optimizer, a part of the database server that formulates the query-execution plan. The optimizer exhaustively considers all possible query plans. The one with the lowest estimated cost is executed by the database server.

The following operations in a query are time-consuming:

- Reading rows from disk sequentially, nonsequentially by ROWID, and nonsequentially through an index
- Reading rows over a network
- Sorting
- Performing some difficult matches of regular expressions
- Indexing and sorting NCHAR and NVARCHAR columns

The general method to speed a query is first to study the query plan of the optimizer as documented by the SET EXPLAIN ON feature, and second to change the query so that the database server performs the following actions:

- Reads fewer pages of data from disk
- Reads pages sequentially rather than nonsequentially
- Avoids sorting or sorts fewer rows or sorts on narrower columns

You can achieve remarkable time savings in this way. Even if no time is saved, the necessary study of the application and the query at least make it clear where the time is being spent.

Creating and Using Stored Procedures

Chapter Overview	3
Introduction to Stored Procedures and SPL	3
What You Can Do with Stored Procedures	3
Relationship Between SQL and a Stored Procedure	4
Creating and Using Stored Procedures	4
Creating a Procedure Using DB-Access	5
Creating a Procedure Using an SQL API	5
Commenting and Documenting a Procedure	6
Diagnosing Compile-Time Errors	6
Finding Syntax Errors in a Procedure Using DB-Access	6
Finding Syntax Errors in a Procedure Using an SQL API	7
Looking at Compile-Time Warnings	8
Generating the Text or Documentation	8
Looking at the Procedure Text	9
Looking at the Procedure Documentation	9
Executing a Procedure	9
Executing a Stored Procedure Dynamically	11
Debugging a Procedure	11
Re-creating a Procedure	13
Privileges on Stored Procedures	13
Privileges Necessary at Creation	14
Privileges Necessary at Execution	14
Owner-Privileged Procedures	15
DBA-Privileged Procedures	15
Privileges and Nested Procedures	16
Revoking Privileges	16

Variables and Expressions	16
Variables	16
Format of Variables	16
Global and Local Variables	17
Defining Variables	17
Data Types for Variables	17
Using Subscripts with Variables	18
Scope of Variables	18
Variable/Keyword Ambiguity	19
SPL Expressions	21
Assigning Values to Variables	21
Program Flow Control	22
Branching	22
Looping	23
Function Handling	24
Calling Procedures Within a Procedure	24
Running an Operating System Command from Within a Procedure	24
Recursively Calling a Procedure	24
Passing Information into and out of a Procedure	25
Returning Results	25
Specifying Return Data Types	25
Returning the Value	26
Returning More Than One Set of Values from a Procedure	26
Exception Handling	27
Trapping an Error and Recovering	28
Scope of Control of an ON EXCEPTION Statement	29
User-Generated Exceptions	30
Simulating SQL Errors	31
Using RAISE EXCEPTION to Exit Nested Code	32
Summary	33

Chapter Overview

You can write procedures using SQL and some additional statements belonging to the Stored Procedure Language (SPL), and store this procedure in the database. These stored procedures are effective tools for controlling SQL activity. This chapter provides instruction on how to write stored procedures. To help you learn how to write them, examples of working stored procedures are provided.

The syntax for each of the SPL statements is described in Chapter 2 of the *Informix Guide to SQL: Syntax*. Accompanying the syntax for each statement are usage notes and examples pertinent to that statement.

Introduction to Stored Procedures and SPL

To SQL, a stored procedure is a user-defined function. Anyone who has the Resource privilege on a database can create a stored procedure. Once the stored procedure is created, it is stored in an executable format in the database as an object of the database. You can use stored procedures to perform any function you can perform in SQL as well as to expand what you can accomplish with SQL alone.

You write a stored procedure using SQL and SPL statements. SPL statements can be used only inside the CREATE PROCEDURE and CREATE PROCEDURE FROM statements. The CREATE PROCEDURE statement is available with **DB-Access**, and the CREATE PROCEDURE and CREATE PROCEDURE FROM statements are available with SQL APIs such as **INFORMIX-ESQL/C** and **INFORMIX-ESQL/COBOL**.

What You Can Do with Stored Procedures

You can accomplish a wide range of objectives with stored procedures, including improving database performance, simplifying the writing of applications, and limiting or monitoring access to data.

Because a stored procedure is stored in an executable format, you can use it to execute frequently repeated tasks to improve performance. Executing a stored procedure rather than straight SQL code allows you to bypass repeated parsing, validity checking, and query optimization.

Because a stored procedure is an object in the database, it is available to every application running on the database. Several applications can use the same stored procedure, so development time for applications is reduced.

You can write a stored procedure to be run with the DBA privilege by a user who does not have the DBA privilege. This allows you to limit and control access to data in the database. Alternatively, a stored procedure can monitor which users access certain tables or data. (See “Controlling Access to Data Using Stored Procedures” on page 11-15 for a discussion of this topic.)

Relationship Between SQL and a Stored Procedure

You can call a procedure within data-manipulation SQL statements and issue SQL statements within a procedure. See Chapter 1 of the *Informix Guide to SQL: Syntax* for a complete list of data manipulation SQL statements.

You use a stored procedure within a data-manipulation SQL statement to supply values to that statement. For example, you can use a procedure to perform the following actions:

- Supply values to be inserted into a table.
- Supply a value that makes up part of a condition clause in a SELECT, DELETE, or UPDATE statement.

These are two possible uses of a procedure within a data manipulation statement, but others exist. In fact, any expression within a data manipulation SQL statement can consist of a procedure call.

You can also issue SQL statements within a stored procedure to hide those SQL statements from a database user. Rather than having all users learn how to use SQL, one experienced SQL user can write a stored procedure to encapsulate an SQL activity, and let others know that the procedure is stored in the database so they can execute it.

Creating and Using Stored Procedures

To write a stored procedure, put the SQL statements that you want run as part of the procedure inside the statement block in a CREATE PROCEDURE statement. You can use SPL statements to control the flow of the operation within the procedure. SPL statements include IF, FOR, and others and are described

in the *Informix Guide to SQL: Syntax*. The CREATE PROCEDURE and CREATE PROCEDURE FROM statements are also described in Chapter 1 of the *Informix Guide to SQL: Syntax*.

Creating a Procedure Using DB-Access

To create a stored procedure using **DB-Access**, issue the CREATE PROCEDURE statement, including all the statements that are part of the procedure in the statement block. For example, to create a procedure that reads a customer address, you can use a statement such as the one in Figure 14-1.

```
CREATE PROCEDURE read_address (lastname CHAR(15)) -- one argument
RETURNING CHAR(15), CHAR(15), CHAR(20), CHAR(15), CHAR(2), CHAR(5); -- 6 items

DEFINE p_lname, p_fname, p_city CHAR(15); --define each procedure variable
DEFINE p_add CHAR(20);
DEFINE p_state CHAR(2);
DEFINE p_zip CHAR(5);

SELECT fname, address1, city, state, zipcode
INTO p_fname, p_add, p_city, p_state, p_zip
FROM customer
WHERE lname = lastname;

RETURN p_fname, lastname, p_add, p_city, p_state, p_zip; --6 items
END PROCEDURE

DOCUMENT 'This procedure takes the last name of a customer as', --brief description
'its only argument. It returns the full name and address of',
'the customer.'
WITH LISTING IN '/acctng/test/listfile' -- compile-time warnings go here
; -- end of the procedure read_address
```

Figure 14-1 Procedure that reads from the customer table

Creating a Procedure Using an SQL API

To create a stored procedure using an SQL API, put the text of the CREATE PROCEDURE statement in a file. Use the CREATE PROCEDURE FROM statement and refer to that file to compile the procedure. For example, to create a procedure to read a customer name, you can use a statement such as the one in Figure 14-1 and store it in a file. If the file is named **read_add_source**, the following statement compiles the **read_address** procedure:

```
CREATE PROCEDURE FROM 'read_add_source';
```

Figure 14-2 shows how the previous SQL statement looks in an ESQL/C program.

```

/* This program creates whatever procedure is in *
 * the file 'read_add_source'.
 */
#include <stdio.h>
#include sqlca;
#include sqlda;
#include datetime;
/* Program to create a procedure from the pwd */

main()
{
  $database play;
  $create procedure from 'read_add_source';
}

```

Figure 14-2 *SQL statement that compiles and stores the read_address procedure in an ESQL/C program*

Commenting and Documenting a Procedure

Observe that the **read_address** procedure in Figure 14-1 includes comments and a DOCUMENT clause. The comments are incorporated into the text of the procedure. Any characters following a double hyphen (--) are considered to be a comment. The double hyphen can be used anywhere within a line.

The text in the DOCUMENT clause should give a summary of the procedure. You can extract this text by querying the **sysprocbody** system catalog table. See “Looking at the Procedure Documentation” on page 14-9 for more information about reading the DOCUMENT clause.

Diagnosing Compile-Time Errors

When you issue a CREATE PROCEDURE or CREATE PROCEDURE FROM statement, the statement fails if a syntax error occurs in the body of the procedure. The database server stops processing the text of the procedure and returns the location of the error.

Finding Syntax Errors in a Procedure Using DB-Access

If a procedure created using **DB-Access** has a syntax error, when you choose the Modify option of the SQL menu, the cursor sits on the line containing the offending syntax.

Finding Syntax Errors in a Procedure Using an SQL API

If a procedure created using an SQL API has a syntax error, the CREATE PROCEDURE statement fails. The database server sets the SQLCODE field of the SQLCA to a negative number and sets the fifth element of the SQLERRD array to the character offset into the file. The particular fields of the SQLCA for each product are shown in the following table:

ESQL/C	ESQL/FORTRAN	ESQL/COBOL
sqlca.sqlcode SQLCODE	sqlcod	SQLCODE OF SQLCA
sqlca.sqlerrd[4]	sqlca.sqlerr(5)	SQLERRD[5] OF SQLCA

Figure 14-3 shows how to trap for a syntax error when you are creating a procedure. It also shows how to display a message and character position where the error occurred.

```
#include <stdio.h>
#include sqlca;
#include sqlda;
#include datetime;
/* Program to create a procedure from procfile in pwd */

main()
{
    long char_num;

    $ database play;
    $create procedure from 'procfile';
    if (sqlca.sqlcode != 0 )
    {
        printf('\nSqlca.sqlcode = %ld\n', sqlca.sqlcode);
        char_num = sqlca.sqlerrd[4];
        printf('\nError in creating read_address. Check character position %ld\n',
            char_num);
    }
    .
    .
    .
}
```

Figure 14-3 Checking for failure when creating a procedure using ESQL/C

In Figure 14-3, if the CREATE PROCEDURE FROM statement fails, the program displays a message in addition to the character position at which the syntax error occurred.

Looking at Compile-Time Warnings

If the database server detects a potential problem but the procedure is syntactically correct, the database server generates a warning and places it in a listing file. You can examine this file to check for potential problems before executing the procedure.

To obtain the listing of compile-time warnings for your procedure, use the `WITH LISTING IN` clause in your `CREATE PROCEDURE` statement, as shown in the following example:

```
CREATE PROCEDURE read_address (lastname CHAR(15)) -- one argument
    RETURNING CHAR(15), CHAR(15), CHAR(20), CHAR(15), CHAR(2), CHAR(5); -- 6 items
.
.
.
    WITH LISTING IN '/acctng/test/listfile' -- compile-time warnings go here
; -- end of the procedure read_address
```

If you are working on a network, the listing file is created on the machine where the database resides. If you provide an absolute pathname and filename for the file, the file is created where you specify. If you provide a relative pathname for the listing file, the file is created in your home directory on the machine where the database resides. (If you do not have a home directory, the file is created in the **root** directory.)

After you create the procedure, you can view the file specified in the `WITH LISTING IN` clause to see the warnings it contains.

Generating the Text or Documentation

Once you create the procedure, it is stored in the **sysprocbody** system catalog table. The **sysprocbody** system catalog table contains the executable procedure as well as the text of the original `CREATE PROCEDURE` statement and the documentation text.

Looking at the Procedure Text

To generate the text of the procedure, select the data column from the **sysprocbody** system catalog table. The following SELECT statement reads the **read_address** procedure text:

```
SELECT data FROM informix.sysprocbody
  WHERE datakey = 'T'      -- find text lines
  AND
  procid = (SELECT procid FROM informix.sysprocedures
            WHERE informix.sysprocedures.procname = 'read_address')
```

Looking at the Procedure Documentation

If you want to view only the documenting text of the procedure, use the following SELECT statement to read the documentation string. The documentation lines found in the following example are those in the DOCUMENT clause of the CREATE PROCEDURE statement.

```
SELECT data FROM informix.sysprocbody
  WHERE datakey = 'D'      -- find documentation lines
  AND
  procid = (SELECT procid FROM informix.sysprocedures
            WHERE informix.sysprocedures.procname = 'read_address')
```

Executing a Procedure

You can execute a procedure in several ways. You can use the SQL statement EXECUTE PROCEDURE or either the LET or CALL SPL statement. In addition, you can dynamically execute procedures, as described in the section, “Executing a Stored Procedure Dynamically” on page 14-11.

The **read_address** procedure returns the full name and address of a customer. To run **read_address** on a customer called “Putnum” using EXECUTE PROCEDURE, enter the following statement:

```
EXECUTE PROCEDURE read_address ('Putnum ');
```

The **read_address** procedure returns values; therefore, if you are executing a procedure from an SQL API or from another procedure, you must use an INTO clause with host variables to receive the data. For example, executing the **read_address** procedure in an ESQL/C program is accomplished with the code segment shown in Figure 14-4.

```
#include <stdio.h>
#include sqlca;
#include sqllda;
#include datetime;
/* Program to execute a procedure in the database named 'play' */

main()
{
    $ char lname[16], fname[16], address[21];
    $ char city[16], state[3], zip[6];

    $ connect to play;
    $EXECUTE PROCEDURE read_address ('Putnum')
        INTO $lname, $fname, $address, $city, $state, $zip;
    if (sqlca.sqlcode != 0 )
        printf('\nFailure on execute');
    }
}
```

Figure 14-4 *Executing a procedure using ESQL/C*

If you are executing a procedure within another procedure, you can use the SPL statements CALL or LET to run the procedure. To use the CALL statement with the **read_address** procedure, you can use the code shown in Figure 14-5.

```
CREATE PROCEDURE address_list ()

    DEFINE p_lname, p_fname, p_city CHAR(15);
    DEFINE p_add CHAR(20);
    DEFINE p_state CHAR(2);
    DEFINE p_zip CHAR(5);
    .
    .
    .
    CALL read_address ('Putnum') RETURNING p_fname, p_lname,
        p_add, p_city, p_state, p_zip;
    .
    .
    .
    -- use the returned data some way
END PROCEDURE;
```

Figure 14-5 *Calling a procedure inside another procedure with the CALL statement*

Figure 14-6 provides an example of using the LET statement to assign values to procedural variables through a procedure call.

```
CREATE PROCEDURE address_list ()  
  
    DEFINE p_lname, p_fname, p_city CHAR(15);  
    DEFINE p_add CHAR(20);  
    DEFINE p_state CHAR(2);  
    DEFINE p_zip CHAR(5);  
    .  
    .  
    LET p_fname, p_lname, p_add, p_city, p_state, p_zip = read_address ('Putnum');  
    .  
    .  
    -- use the returned data some way  
END PROCEDURE;
```

Figure 14-6 *Assigning values from a procedure call with a LET statement*

Executing a Stored Procedure Dynamically

You can prepare an EXECUTE PROCEDURE statement in conjunction with the ALLOCATE DESCRIPTOR and GET DESCRIPTOR statements in an ESQL/C program. Parameters to the stored procedure can be passed in the same manner as the SELECT statement and may be passed at runtime or compile time. For a detailed example of executing a stored procedure dynamically, see Chapter 10 of the *INFORMIX-ESQL/C Programmer's Manual*. For information about dynamic SQL and using a prepared SELECT statement, see Chapter 5, "Programming with SQL."

Debugging a Procedure

Once you successfully create and run a procedure, you can encounter logical errors. If logical errors are in the procedure, use the TRACE statement to help find them. You can trace the values of the following procedural entities:

- Variables
- Procedure arguments
- Return values
- SQL error codes
- ISAM error codes

To generate a listing of traced values, first use the SQL statement SET DEBUG FILE to name the file that is to contain the traced output. When you create your procedure, include the TRACE statement in one of its forms.

The following ways specify the form of TRACE output:

TRACE ON	traces all statements except SQL statements. The contents of variables are printed before being used. Procedure calls and returned values are also traced.
TRACE PROCEDURE	traces only the procedure calls and returned values.
TRACE <i>expression</i>	prints a literal or an expression. If necessary, the value of the expression is calculated before being sent to the file.

Figure 14-7 shows how you can use the TRACE statement with a version of the **read_address** procedure. Several of the SPL statements shown in this example have not been discussed, but the entire example demonstrates how the TRACE statement can help you monitor execution of the procedure.

```

CREATE PROCEDURE read_many (lastname CHAR(15))
    RETURNING CHAR(15), CHAR(15), CHAR(20), CHAR(15),CHAR(2), CHAR(5);

    DEFINE p_lname,p_fname, p_city CHAR(15);
    DEFINE p_add CHAR(20);
    DEFINE p_state CHAR(2);
    DEFINE p_zip CHAR(5);
    DEFINE lcount, i INT;

    LET lcount = 1;

    TRACE ON; -- Every expression will be traced from here on
    TRACE 'Foreach starts'; -- A trace statement with a literal
    FOREACH
    SELECT fname, lname, address1, city, state, zipcode
        INTO p_fname, p_lname, p_add, p_city, p_state, p_zip
        FROM customer
        WHERE lname = lastname
    RETURN p_fname, p_lname, p_add, p_city, p_state, p_zip WITH RESUME;
    LET lcount = lcount + 1; -- count of returned addresses
    END FOREACH;

    TRACE 'Loop starts'; -- Another literal
    FOR i IN (1 TO 5)
    BEGIN
        RETURN i , i+1, i*i, i/i, i-1,i with resume;
    END
    END FOR;

END PROCEDURE

;

```

Figure 14-7 *Using the TRACE statement to monitor execution of a procedure*

Each time you execute the traced procedure, entries are added to the file you specified using the SET DEBUG FILE statement. To see the debug entries, view the output file using any text editor.

The following list contains some of the output generated by the procedure in Figure 14-7. Next to each traced statement is an explanation of its contents.

<code>TRACE ON</code>	echoed TRACE ON statement.
<code>TRACE Foreach starts</code>	traced expression, in this case, the literal string <code>Foreach starts</code> .
<code>start select cursor</code>	notification that a cursor is opened to handle a FOREACH loop.
<code>select cursor iteration</code>	notification of the start of each iteration of the select cursor.
<code>expression: (+lcount, 1)</code>	the encountered expression, <code>(lcount+1)</code> evaluates to 2.
<code>let lcount = 2</code>	each LET statement is echoed with the value.

Re-creating a Procedure

If a procedure exists in a database, you must drop it explicitly using the DROP PROCEDURE statement before you can create another procedure with the same name. If you debug your procedure and attempt to use the CREATE PROCEDURE statement with the same procedure name again, the attempt fails unless you first drop the existing procedure from the database.

Privileges on Stored Procedures

A stored procedure resides in the database in which it was created. As with other database objects, you need appropriate privileges to create a stored procedure. In addition, you need appropriate privileges to execute a stored procedure.

Two types of stored procedures exist in a database: DBA-privileged and owner-privileged. When you create the procedure, you specify which type it is. You need different privileges to create and execute these two types of procedures. The differences are described in the sections that follow and are summarized in Figure 14-8.

	DBA-privileged procedure	Owner-privileged procedure
Can be created by:	Any user with the DBA privilege	Any user with at least the Resource privilege
Users having Execute privilege by default:	Any user with the DBA privilege	Not ANSI-compliant: Public (any user with Connect database privilege) ANSI-compliant: The procedure owner and any user with the DBA privilege
Privileges the procedure owner or DBA must grant another user to enable that user to run a procedure:	Execute privilege	Execute privilege and privileges on underlying objects. If owner has privileges on underlying objects with the GRANT WITH option, only the Execute privilege is required

Figure 14-8 *Differences between DBA-privileged and owner-privileged procedures*

Privileges Necessary at Creation

Only users who have the DBA privilege can create a DBA-privileged procedure. To create an owner-privileged procedure, you need to have at least the Resource privilege. See “Granting and Limiting Access to Your Database” on page 11-3.

Privileges Necessary at Execution

To run a procedure you always need the Execute privilege for that procedure or DBA database privileges. Depending on whether the procedure is a DBA-mode procedure and if the database is ANSI-compliant, certain privileges are granted to users implicitly by the database server.

If the procedure is owner-privileged, the database server grants the Execute privilege to PUBLIC. If the database is ANSI-compliant, the database server grants only the Execute privilege to the owner and users with DBA status.

If the procedure is DBA-privileged, the database server grants the Execute privilege to all users having the DBA privilege.

Owner-Privileged Procedures

When you execute an owner-privileged procedure, the database server checks the existence of any referenced objects. In addition, the database server verifies that you have the necessary privileges on the referenced objects.

If you execute a procedure that only references objects that you own, no privilege conflicts occurs. If you do not own the referenced objects and you execute a procedure that contains, say, `SELECT` statements, for example, you need the `Select` privilege on the appropriate tables to run the procedure without generating errors.

If the owner has the necessary privileges with the `WITH GRANT` option, those privileges are automatically conferred to you when the owner issues a `GRANT EXECUTE` statement.

Unqualified objects created in the course of the procedure are owned by the owner of the procedure, not the user running the procedure. For example, the following example shows lines in an owner-privileged stored procedure that create two tables. If this procedure is owned by **tony** and a user **marty** runs the procedure, the first table, **gargantuan**, is owned by **tony**. The second table, **tiny**, is owned by **libby**. The table **gargantuan** is an unqualified name; therefore, **tony** owns the table **gargantuan**. The table **tiny** is qualified by the owner **libby**, so **libby** owns the table **tiny**.

```
CREATE PROCEDURE tryit()  
.  
.  
.  
CREATE TABLE gargantuan (col1 INT, col2 INT, col3 INT);  
CREATE TABLE libby.tiny (col1 INT, col2 INT, col3 INT);  
  
END PROCEDURE
```

DBA-Privileged Procedures

When you execute a DBA-privileged procedure, you assume the privileges of a DBA for the duration of the procedure. A DBA-privileged procedure acts as if the user running the procedure is first granted DBA privilege, then executes each statement of the procedure manually, and finally has DBA privilege revoked.

Objects created in the course of running a DBA procedure are owned by the user running the procedure, unless the data-definition statement in the procedure explicitly names the owner to be someone else.

Privileges and Nested Procedures

DBA-privileged status is not inherited by a called procedure. For example, if a DBA-privileged procedure executes an owner-privileged procedure, the owner-privileged procedure does not run as a DBA procedure. If an owner-privileged procedure calls a DBA-privileged procedure, the statements within the DBA-privileged procedure execute as they would within any DBA-privileged procedure.

Revoking Privileges

The owner of a procedure can revoke Execute privilege from a user. If a user loses the Execute privilege on a procedure, the Execute privilege also is revoked from all users who were granted the Execute privilege by that user.

Variables and Expressions

This section discusses how to define and use variables in SPL. The differences between SPL and SQL expressions also are covered here.

Variables

You can use a variable in a stored procedure in several ways. You can use a variable in a database query or other SQL statement wherever a constant is expected. You can use a variable with SPL statements to assign and calculate values, keep track of the number of rows returned from a query, and execute a loop as well as handle other tasks.

The value of a variable is held in memory; the variable is not a database object. Hence, rolling back a transaction does not restore values of procedural variables.

Format of Variables

A variable follows the rules of an SQL identifier. (See Chapter 1 of the *Informix Guide to SQL: Syntax*.) Once you define a variable, you can use it anywhere in the procedure as appropriate.

If you are using an SQL API, you do not have to set off the variable with a special symbol (unlike host variables in an SQL API).

Global and Local Variables

You can define a variable to be either local or global. A variable is local by default. The following outline provides the differences between the two types:

Local	A local variable is available only within the procedure in which it is defined. Local variables do not allow a default value to be assigned at compile time.
Global	<p>A global variable is available to other procedures run by the same user session in the same database. The values of global variables are stored in memory. The global environment is the memory used by all the procedures run within a given session on a given database server, such as all procedures run by an SQL API or in a DB-Access session. The values of the variables are lost when the session ends.</p> <p>Global variables require a default value to be assigned at compile time.</p> <p>The first definition of a global variable puts the variable into the global environment. Subsequent definitions of the same variable, in different procedures, simply bind the variable to the global environment.</p>

Defining Variables

You define variables using the **DEFINE** statement. If you list a variable in the argument list of a procedure, the variable is defined implicitly, and you do not need to define it formally with the **DEFINE** statement. You must assign a value, which may be null, to a variable before you can use it.

Data Types for Variables

You can define a variable as any of the data types available for columns in a table except **SERIAL**. The following example shows several cases of defined procedural variables:

```
DEFINE x INT;
DEFINE name CHAR(15);
DEFINE this_day DATETIME YEAR TO DAY ;
```

If you define a variable for **TEXT** or **BYTE** data, the variable does not actually contain the data; instead, it serves as a pointer to the data. However, you use this procedural variable as you would use any other procedural variable.

When you define a TEXT or BYTE variable, you must use the word REFERENCES, which emphasizes that these variables do not contain the data; they simply reference the data. The following example shows the definition of a TEXT and a BYTE variable:

```
DEFINE ttt REFERENCES TEXT;  
DEFINE bbb REFERENCES BYTE;
```

Using Subscripts with Variables

You can use subscripts with variables having CHAR, VARCHAR, NCHAR, NVARCHAR, BYTE or TEXT data types, just as you can with SQL column names. The subscripts indicate the starting and ending character positions of the variable. Subscripts must always be constants. You cannot use variables as subscripts. The following example illustrates the usage:

```
DEFINE name CHAR(15);  
LET name[4,7] = 'Ream';  
SELECT fname[1,3] INTO name[1,3] FROM customer  
WHERE lname = 'Ream';
```

The portion of the variable contents that is delimited by the two subscripts is referred to as a substring.

Scope of Variables

A variable is valid within the statement block in which it is defined. It is valid within statement blocks nested within that statement block as well, unless it is masked by a redefinition of a variable with the same name.

In the beginning of the procedure in Figure 14-9, the integer variables *x*, *y*, and *z* are defined and initialized. The BEGIN and END statements mark a nested statement block in which the integer variables *x* and *q* are defined, as

well as the CHAR variable `z`. Within the nested block, the redefined variable `x` masks the original variable `x`. After the `END` statement, which marks the end of the nested block, the original value of `x` is accessible again.

```
CREATE PROCEDURE scope()  
  DEFINE x,y,z INT;  
  LET x = 5; LET y = 10;  
  LET z = x + y; --z is 15  
  BEGIN  
    DEFINE x, q INT; DEFINE z CHAR(5);  
    LET x = 100;  
    LET q = x + y; -- q = 110  
    LET z = 'silly'; -- z receives a character value  
  END  
  LET y = x; -- y is now 5  
  LET x = z; -- z is now 15, not 'silly'  
END PROCEDURE
```

Figure 14-9 *Masking of variables with nested statement blocks*

Variable/Keyword Ambiguity

If you define a variable as a keyword, ambiguities can occur. The following rules for identifiers help you avoid ambiguities for variables, procedure names, and system function names:

- Defined variables take the highest precedence.
- Procedures defined as such in a `DEFINE` statement take precedence over SQL functions.
- SQL functions take precedence over procedures that exist but are *not* identified as procedures in a `DEFINE` statement.

In some cases, you must change the name of the variable. For example, you cannot define a variable with the name **count** or **max**, because these are the names of aggregate functions. Refer to Chapter 1 of the *Informix Guide to SQL: Syntax* for a list of the keywords that can be used ambiguously.

Variables and Column Names

If you use the same identifier for a procedural variable as you use for a column name, the database server assumes that each instance of the identifier is a variable. Qualify the column name with the table name to use the identifier

as a column name. In the following example, the procedural variable **lname** is the same as the column name. In the following SELECT statement, **customer.lname** is a column name and **lname** is a variable name:

```
CREATE PROCEDURE table_test()

    DEFINE lname CHAR(15);
    LET lname = 'Miller';
.
.
.
    SELECT customer.lname FROM customer INTO lname
        WHERE customer_num = 502;
.
.
.
```

Variables and SQL Functions

If you use the same identifier for a procedural variable as for an SQL function, the database server assumes that an instance of the identifier is a variable and disallows the use of the SQL function. You cannot use the SQL function within the block of code in which the variable is defined. The following example shows a block within a procedure in which the variable called **user** is defined. This definition disallows the use of the USER function while in the BEGIN...END block.

```
CREATE PROCEDURE user_test()
    DEFINE name CHAR(10);
    DEFINE name2 CHAR(10);
    LET name = user; -- the SQL function

    BEGIN
        DEFINE user CHAR(15); -- disables user function
        LET user = 'Miller';
        LET name = user; -- assigns 'Miller' to variable name

    END
.
.
.
    LET name2 = user; -- SQL function again
```

Procedure Names and SQL Functions

For information about ambiguities between procedure names and SQL function names, see the *Informix Guide to SQL: Syntax*.

SPL Expressions

You can use any SQL expression in a stored procedure except for an aggregate expression. The complete syntax and notes for SQL expressions are described in Chapter 1 of the *Informix Guide to SQL: Syntax*.

The following examples contain SQL expressions:

```
var1
var1 + var2 + 5
read_address('Miller')
read_address(lastname = 'Miller')
get_duedate(acct_num) + 10 UNITS DAY
fname[1,5] || ' ' || lname
'(415)' || get_phonenum(cust_name)
```

Assigning Values to Variables

The following ways can assign a value to a procedure variable:

- Using a LET statement
- Using a SELECT...INTO statement
- Using a CALL statement with a procedure that has a RETURNING clause
- Using an EXECUTE PROCEDURE...INTO statement

Use the LET statement to assign a value to one or more variables. The following example illustrates several forms of the LET statement:

```
LET a = b + a;
LET a, b = c, d;
LET a, b = (SELECT fname, lname FROM customer
           WHERE customer_num = 101);
LET a, b = read_name(101);
```

Use the SELECT statement to assign a value directly from the database to a variable. The statement in the following example accomplishes the same task as the third LET statement in the previous example:

```
SELECT fname, lname into a, b FROM customer
WHERE customer_num = 101
```

Use the CALL or EXECUTE PROCEDURE statements to assign values returned by a procedure to one or more procedural variables. Both statements in the following example return the full address from the procedure **read_address** into the specified procedural variables:

```
EXECUTE PROCEDURE read_address('Smith')
  INTO p_fname, p_lname, p_add, p_city, p_state, p_zip;

CALL read_address('Smith')
  RETURNING p_fname, p_lname, p_add, p_city, p_state, p_zip;
```

Program Flow Control

SPL contains several statements that enable you to control the flow of your stored procedure and to make decisions based on data obtained at run time. These program-flow-control statements are described briefly in this section. Their syntax and complete descriptions are provided in Chapter 2 of the *Informix Guide to SQL: Syntax*.

Branching

Use an IF statement to form a logic branch in a stored procedure. An IF statement first evaluates a condition and, if the condition is true, the statement block contained in the THEN portion of the statement is executed.

If the condition is not true, execution falls through to the next statement, unless the IF statement includes an ELSE clause or ELIF (else if) clause. Figure 14-10 shows an example of an IF statement.

```
CREATE PROCEDURE str_compare (str1 CHAR(20), str2 CHAR(20))
  RETURNING INT;
  DEFINE result INT;

  IF str1 > str2 THEN
    result = 1;
  ELIF str2 > str1 THEN
    result = -1;
  ELSE
    result = 0;
  END IF
  RETURN result;
END PROCEDURE -- str_compare
```

Figure 14-10 Using an IF statement

Looping

Three methods exist of looping in SPL, accomplished with one of the following statements:

- | | |
|---------|---------------------------------------------------------------------------------------------------------------------|
| FOR | initiates a controlled loop. Termination is guaranteed. |
| FOREACH | allows you to select and manipulate more than one row from the database. It declares and opens a cursor implicitly. |
| WHILE | initiates a loop. Termination is <i>not</i> guaranteed. |

Four ways exist to leave a loop, which can be accomplished with one of the following statements:

- | | |
|-----------------|------------------------------------------------------------------------------------------------------------|
| CONTINUE | skips the remaining statements in the present, identified loop and starts the next iteration of that loop. |
| EXIT | exits the present, identified loop. Execution resumes at the first statement after the loop. |
| RETURN | exits the procedure. If a return value is specified, that value is returned upon exit. |
| RAISE EXCEPTION | exits the loop if the exception is not trapped (caught) in the body of the loop. |

See Chapter 2 of the *Informix Guide to SQL: Syntax* for more information concerning the syntax and use of these statements.

Function Handling

You can call procedures as well as run operating system commands from within a procedure.

Calling Procedures Within a Procedure

Use a CALL statement or the SQL statement EXECUTE PROCEDURE to execute a procedure from a procedure. Figure 14-11 shows a call to the **read_name** procedure using a CALL statement.

```
CREATE PROCEDURE call_test()
    RETURNING CHAR(15), CHAR(15);

    DEFINE fname, lname CHAR(15);
    CALL read_name('Putnum') RETURNING fname, lname;

    IF fname = 'Eileen' THEN RETURN 'Jessica', lname;
    ELSE RETURN fname, lname;
    END IF
END PROCEDURE
```

Figure 14-11 *Calling a procedure with a CALL statement*

Running an Operating System Command from Within a Procedure

Use the SYSTEM statement to execute a system call from a procedure. Figure 14-12 shows a call to the **echo** command.

```
CREATE DBA PROCEDURE delete_customer(cnum INT)

DEFINE username CHAR(8);

DELETE FROM customer
    WHERE customer_num = cnum;

IF username = 'acctclrk' THEN
    SYSTEM 'echo ''Delete from customer by acctclrk'' >> /mis/records/updates' ;
END IF
END PROCEDURE -- delete_customer
```

Figure 14-12 *Making a system call from a procedure with a SYSTEM statement*

Recursively Calling a Procedure

You can call a procedure from itself. No restrictions apply on calling a procedure recursively.

Passing Information into and out of a Procedure

When you create a procedure, you determine whether it expects information to be passed to it by specifying an argument list. For each piece of information that the procedure expects, you specify one argument and the data type of that argument.

For example, if a procedure needs to have a single piece of integer information passed to it, you can provide a procedure heading as shown in the following example:

```
CREATE PROCEDURE safe_delete(cnum INT)
```

Returning Results

A procedure that returns one or more values must contain two lines of code to accomplish the transfer of information: one line to state the data types that are going to be returned and one line to return the values explicitly.

Specifying Return Data Types

Immediately after you specify the name and input parameters of your procedure, you must include a RETURNING clause with the data type of each value you expect to be returned. The following example shows the header of a procedure (name, parameters, and RETURNING clause) that expects one integer as input and returns one integer and one 10-byte character value:

```
CREATE PROCEDURE get_call(cnum INT)
    RETURNING INT, CHAR(10);
```

Returning the Value

Once you use the RETURNING clause to indicate the type of values that are to be returned, you can use the RETURN statement at any point in your procedure to return the same number and data types as listed in the RETURNING clause. The following example shows how you can return information from the `get_call` procedure:

```
CREATE PROCEDURE get_call(cnum INT)
  RETURNING INT, CHAR(10);
  DEFINE ncalls INT;
  DEFINE o_name CHAR(10);
  .
  .
  .
  RETURN ncalls, o_name;
  .
  .
  .
END PROCEDURE
```

If you neglect to include a RETURN statement, you do not get an error message at compile time.

Returning More Than One Set of Values from a Procedure

If your procedure performs a select that can return more than one row from the database or if you return values from inside a loop, you must use the WITH RESUME keywords in the RETURN statement. Using a RETURN...WITH RESUME statement causes the value or values to be returned to the calling program or procedure. After the calling program receives the values, execution returns to the statement immediately following the RETURN...WITH RESUME statement.

Figure 14-13 is an example of a *cursory* procedure. It returns values from a FOREACH loop and a FOR loop. This procedure is called a cursory procedure because it contains a FOREACH loop.

```

CREATE PROCEDURE read_many (lastname CHAR(15))
    RETURNING CHAR(15), CHAR(15), CHAR(20), CHAR(15),CHAR(2),
    CHAR(5);

    DEFINE p_lname,p_fname, p_city CHAR(15);
    DEFINE p_add CHAR(20);
    DEFINE p_state CHAR(2);
    DEFINE p_zip CHAR(5);
    DEFINE lcount INT ;
    DEFINE i INT ;

    LET lcount = 0;
    TRACE ON;
    CREATE VIEW myview AS SELECT * FROM customer;
    TRACE 'Foreach starts';
    FOREACH
    SELECT fname, lname, address1, city, state, zipcode
        INTO p_fname, p_lname, p_add, p_city, p_state, p_zip
        FROM customer
        WHERE lname = lastname
    RETURN p_fname, p_lname, p_add, p_city, p_state, p_zip
        WITH RESUME;
    LET lcount = lcount +1;
    END FOREACH;

    FOR i IN (1 TO 5)
    BEGIN
        RETURN 'a', 'b', 'c', 'd', 'e' WITH RESUME;
    END
    END FOR;
END PROCEDURE

```

Figure 14-13 *Procedure that returns values from a FOREACH loop and a FOR loop*

When you execute this procedure, it returns the name and address for each person with the specified last name. It also returns a sequence of letters. The calling procedure or program must be expecting multiple returned values, and it must use a cursor or a FOREACH statement to handle the multiple returned values.

Exception Handling

You can trap any exception (or error) returned by the database server to your procedure, or raised by your procedure, with the ON EXCEPTION statement. The RAISE EXCEPTION statement enables you to generate an exception within your procedure.

Trapping an Error and Recovering

The ON EXCEPTION statement provides a mechanism to trap any error.

To trap an error, enclose a group of statements in a statement block and precede the statement block with an ON EXCEPTION statement. If an error occurs in the block that follows the ON EXCEPTION statement, you can take recovery action.

Figure 14-14 shows an example of an ON EXCEPTION statement within a BEGIN...END block.

```
BEGIN
  DEFINE c INT;
  ON EXCEPTION IN
  (
    -206, -- table does not exist
    -217 -- column does not exist
  ) SET err_num

  IF err_num = -206 THEN
    CREATE TABLE t (c INT);
    INSERT INTO t VALUES (10);
    -- continue after the insert statement
  ELSE
    ALTER TABLE t ADD(d INT);
    LET c = (SELECT d FROM t);
    -- continue after the select statement.
  END IF
  END EXCEPTION WITH RESUME

INSERT INTO t VALUES (10); -- will fail if t does not exist

LET c = (SELECT d FROM t); -- will fail if d does not exist
END
```

Figure 14-14 An ON EXCEPTION statement within a BEGIN...END block

When an error occurs, the SPL interpreter searches for the innermost ON EXCEPTION declaration that traps the error. Note that the first action after trapping the error is to reset the error. When execution of the error action code is complete, and if the ON EXCEPTION declaration that was raised included the WITH RESUME keywords, execution resumes automatically with the statement *following* the statement that generated the error. If the ON EXCEPTION declaration did not include the WITH RESUME keywords, execution exits the current block completely.

Scope of Control of an ON EXCEPTION Statement

An ON EXCEPTION statement is valid for the statement block that follows the ON EXCEPTION statement, all of the statement blocks nested within that following statement block, and all of the statement blocks that follow the ON EXCEPTION statement. It is *not* valid in the statement block that contains the ON EXCEPTION statement.

The pseudocode in Figure 14-15 shows where the exception is valid within the procedure. That is, if error 201 occurs in any of the indicated blocks, the action labeled a201 occurs.

```
CREATE PROCEDURE scope()  
  DEFINE i INT;  
  .  
  .  
  .  
  BEGIN  -- begin statement block A  
  .  
  .  
  .  
    ON EXCEPTION IN (201)  
    -- do action a201  
    END EXCEPTION  
    BEGIN -- statement block aa  
      -- do action, a201 valid here  
    END  
    BEGIN -- statement block bb  
      -- do action, a201 valid here  
    END  
    WHILE i < 10  
      -- do something, a201 is valid here  
    END WHILE  
  
  END  
  BEGIN  -- begin statement block B  
    -- do something  
    -- a201 is NOT valid here  
  END  
END PROCEDURE
```

Figure 14-15 Pseudocode showing the *ON EXCEPTION* statement is valid within a procedure

User-Generated Exceptions

You can generate your own error using the `RAISE EXCEPTION` statement, as shown in the following pseudocode example. In this example, the `ON EXCEPTION` statement uses two variables, **esql** and **eisam**, to hold the error numbers returned by the database server. If an error occurs and the SQL error

number is -206, the action defined in the IF clause is taken. If any other SQL error is caught, it is passed out of this BEGIN...END block to the block that contains this block.

```
BEGIN
  ON EXCEPTION SET esql, eisam -- trap all errors
  IF esql = -206 THEN          -- table not found
    -- recover somehow
  ELSE
    RAISE exception esql, eisam ; -- pass the error up
  END IF
END EXCEPTION
  -- do something
END
```

Simulating SQL Errors

You can generate errors to simulate SQL errors, as shown in the following example. Here, if the user is **pault**, then the stored procedure acts as if that user has no update privileges, even if he really does have that privilege.

```
BEGIN
  IF user = 'pault' THEN
    RAISE EXCEPTION -273; -- deny Paul update privilege
  END IF
END
```

Using RAISE EXCEPTION to Exit Nested Code

You can use the RAISE EXCEPTION statement to break out of a deeply nested block, as shown in Figure 14-16. If the innermost condition is true (if aa is negative), then the exception is raised and execution jumps to the code following the END of the block. In this case, execution jumps to the TRACE statement.

```
BEGIN
  ON EXCEPTION IN (1)
  END EXCEPTION WITH RESUME -- do nothing significant (cont)

  BEGIN
    FOR i IN (1 TO 1000)
      FOREACH select ..INTO aa FROM t
        IF aa < 0 THEN
          RAISE EXCEPTION 1 ;    -- emergency exit
        END IF
      END FOREACH
    END FOR
    RETURN 1;
  END

  --do something;                -- emergency exit to
                                -- this statement.

  TRACE 'Negative value returned'
  RETURN -10;
END
```

Figure 14-16 *Breaking out of nested loop with a RAISE EXCEPTION statement*

Remember that a BEGIN...END block is a *single* statement. When an error occurs somewhere inside a block and the trap is outside the block, when execution resumes, the rest of the block is skipped and execution resumes at the next statement.

Unless you set a trap for this error somewhere in the block, the error condition is passed back to the block that contains the call and back to any blocks that contain the block. If no ON EXCEPTION statement exists that is set to handle the error, execution of the procedure stops, creating an error for the program or procedure that is executing the procedure.

Summary

Stored procedures provide many opportunities for streamlining your database process, including enhanced database performance, simplified applications, and limited or monitored access to data. See the *Informix Guide to SQL: Syntax* for syntax diagrams of SPL statements.

Summary

Creating and Using Triggers

Chapter Overview	3
When to Use Triggers	3
How to Create a Trigger	4
Assigning a Trigger Name	5
Specifying the Trigger Event	5
Defining the Triggered Actions	5
A Complete CREATE TRIGGER Statement	6
Using Triggered Actions	7
Using BEFORE and AFTER Triggered Actions	7
Using FOR EACH ROW Triggered Actions	8
Using the REFERENCING Clause	9
Using the WHEN Condition	10
Using Stored Procedures as Triggered Actions	10
Passing Data to a Stored Procedure	11
Using the SPL	11
Updating Nontriggering Columns with Data from a Stored Procedure	12
Tracing Triggered Actions	12
Generating Error Messages	14
Applying a Fixed Error Message	15
Generating a Variable Error Message	16
Summary	17



Chapter Overview

An SQL trigger is a mechanism that resides in the database. It is available to any user who has permission to use it. It specifies that when a particular action—an insert, a delete, or an update—occurs on a particular table, the database server should automatically perform one or more additional actions. The additional actions can be INSERT, DELETE, UPDATE, or EXECUTE PROCEDURE statements.

This chapter describes the purpose of each component of the CREATE TRIGGER statement, illustrates some uses for triggers, and describes the advantages of using a stored procedure as a triggered action.

When to Use Triggers

Because a trigger resides in the database and anyone who has the required privilege can use it, a trigger enables you to write a set of SQL statements that multiple applications can use. It enables you to prevent redundant code when multiple programs need to perform the same database operation.

Among other uses, you can use triggers to perform the following actions:

- Create an audit trail of activity in the database. For example, you can track updates to the orders table by updating corroborating information to an audit table.
- Implement a business rule. For example, you can determine when an order exceeds a customer's credit limit and display a message to that effect.
- Derive additional data that is not available within a table or within the database. For example, when an update occurs to the quantity column of the items table, you can calculate the corresponding adjustment to the total_price column.
- Enforce referential integrity. When you delete a customer, for example, you can use a trigger to delete corresponding rows (that is, rows having the same customer number) in the **orders** table.

How to Create a Trigger

You use the CREATE TRIGGER statement to create a trigger. The CREATE TRIGGER statement is a data definition statement that associates SQL statements with a precipitating action on a table. When the precipitating action occurs, the associated SQL statements, which are stored in the database, are triggered. The following diagram illustrates the relationship of the precipitating action, or trigger event, to the triggered action.

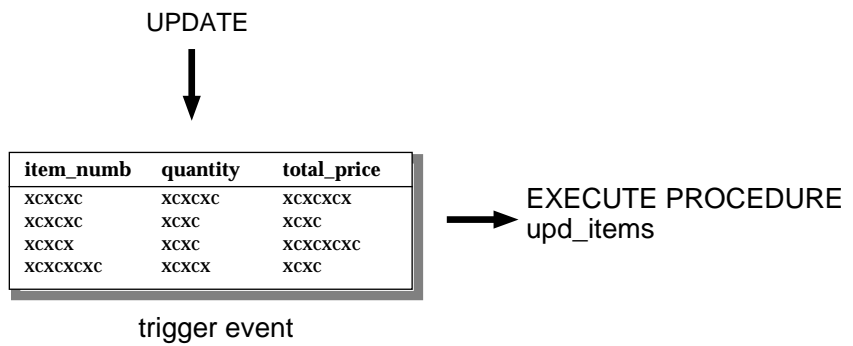


Figure 15-1 A trigger consists of a trigger event and a triggered action

The CREATE TRIGGER statement consists of clauses that perform the following actions:

- Assign a trigger name.
- Specify the trigger event— that is, the table and the type of action that initiate the trigger.
- Define the SQL actions that are triggered.

An optional clause, called the REFERENCING clause is discussed later under “Using FOR EACH ROW Triggered Actions” on page 15-8.

You can create a trigger using the **DB-Access** utility or one of the SQL APIs, **INFORMIX-ESQL/C** or **INFORMIX-ESQL/COBOL**. This section describes the CREATE TRIGGER statement as you would enter it using the interactive Query-language option in **DB-Access**. In an SQL API, you simply precede the statement with the symbol or keywords that identify it as an embedded statement.

Assigning a Trigger Name

The trigger name identifies the trigger. It follows the words `CREATE TRIGGER` in the statement. It can be up to 18 characters inclusive, in length, beginning with a letter and consisting of letters, the digits 0-9, and the underscore. In the following example, the portion of the `CREATE TRIGGER` statement that is shown assigns the name **upqty** to the trigger:

```
CREATE TRIGGER upqty          -- assign trigger name
```

Specifying the Trigger Event

The *trigger event* is the type of statement that activates the trigger. When a statement of this type is performed on the table, the database server executes the SQL statements that make up the triggered action. The trigger event can be an `INSERT`, `DELETE`, or `UPDATE` statement. When you define an `UPDATE` trigger event, you can name one or more columns in the table to activate the trigger. If you do not name any columns, then an update of any column in the table activates the trigger. You can create only one `INSERT` and one `DELETE` trigger per table, but you can create multiple `UPDATE` triggers as long as the triggering columns are mutually exclusive.

In the following excerpt of a `CREATE TRIGGER` statement, the trigger event is defined as an update of the **quantity** column in the **items** table:

```
CREATE TRIGGER upqty
UPDATE OF quantity ON items      -- an UPDATE trigger event
```

Notice that this portion of the statement identifies the table on which you create the trigger. If the trigger event is an insert or delete, only the type of statement and the table name are required, as shown in the following example:

```
CREATE TRIGGER ins_qty
INSERT ON items                  -- an INSERT trigger event
```

Defining the Triggered Actions

The *triggered actions* are the SQL statements that are performed when the trigger event occurs. The triggered actions can consist of INSERT, DELETE, UPDATE, and EXECUTE PROCEDURE statements. In addition to specifying what actions are to be performed, however, you must also specify *when* they are to be performed in relation to the triggering statement. You have the following choices:

- Before the triggering statement executes
- After the triggering statement executes
- For each row that is affected by the triggering statement

A single trigger can define actions for each of these times.

You define a triggered action by specifying when it occurs and then providing the SQL statement or statements to execute. You specify when the action is to occur with the keywords BEFORE, AFTER, or FOR EACH ROW. The triggered actions follow, enclosed in parentheses. The following triggered action definition specifies that the stored procedure **upd_items_p1** is to be executed before the triggering statement:

```
BEFORE(EXECUTE PROCEDURE upd_items_p1) -- a BEFORE action
```

A Complete CREATE TRIGGER Statement

If you combine the trigger name clause, the trigger event clause, and the triggered action clause, you have a complete CREATE TRIGGER statement. The following CREATE TRIGGER statement is the result of combining the components of the statement from the preceding examples. This trigger executes the stored procedure **upd_items_p1** whenever the **quantity** column of the **items** table is updated.

```
CREATE TRIGGER upqty  
UPDATE OF quantity ON items  
BEFORE(EXECUTE PROCEDURE upd_items_p1)
```

Note that if a database object in the trigger definition, such as the stored procedure **upd_items_p1** in this example, does not exist when the database server processes the CREATE TRIGGER statement, it returns an error.

Using Triggered Actions

To use triggers effectively, you need to understand the relationship between the triggering statement and the resulting triggered actions. You define this relationship when you specify the time that the triggered action occurs—that is, BEFORE, AFTER, or FOR EACH ROW.

Using BEFORE and AFTER Triggered Actions

Triggered actions that occur before or after the trigger event execute only once. A BEFORE triggered action executes before the *triggering statement*—that is, before the occurrence of the trigger event. An AFTER triggered action executes after the action of the triggering statement is complete. BEFORE and AFTER triggered actions execute even if the triggering statement does not process any rows.

Among other uses, you can use BEFORE and AFTER triggered actions to determine the effect of the triggering statement. For example, before updating the **quantity** column in the **items** table, you could call the stored procedure **upd_items_p1**, shown in the following example, to calculate the total quantity on order for all items in the table. Notice that the procedure stores the total in a global variable called **old_qty**.

```
CREATE PROCEDURE upd_items_p1()
  DEFINE GLOBAL old_qty INT DEFAULT 0;
  LET old_qty = (SELECT SUM(quantity) FROM items);
END PROCEDURE;
```

After the triggering update completes, you can calculate the total again to see how much it has changed. The following stored procedure, **upd_items_p2**, calculates the total of **quantity** again and stores the result in the local variable **new_qty**. Then it compares **new_qty** to the global variable **old_qty** to see if

the total quantity for all orders has increased by more than 50 percent. If so the procedure uses the `RAISE EXCEPTION` statement to simulate an SQL error.

```
CREATE PROCEDURE upd_items_p2()
  DEFINE GLOBAL old_qty INT DEFAULT 0;
  DEFINE new_qty INT;
  LET new_qty = (SELECT SUM(quantity) FROM items);
  IF new_qty > old_qty * 1.50 THEN
    RAISE EXCEPTION -746, 0, 'Not allowed - rule violation;'
  END IF
END PROCEDURE;
```

The following trigger calls **upd_items_p1** and **upd_items_p2** to prevent an extraordinary update on the **quantity** column of the **items** table:

```
CREATE TRIGGER up_items
UPDATE OF quantity ON items
BEFORE(EXECUTE PROCEDURE upd_items_p1())
AFTER(EXECUTE PROCEDURE upd_items_p2());
```

If an update raises the total quantity on order for all items by more than 50 percent, the `RAISE EXCEPTION` statement in **upd_items_p2** terminates the trigger with an error. When a trigger fails in **INFORMIX-OnLine Dynamic Server** and the database has logging, the database server rolls back the changes made by both the triggering statement and the triggered actions. See `CREATE TRIGGER` in Chapter 1 of the *Informix Guide to SQL: Syntax* for more information on what happens when a trigger fails.

Using FOR EACH ROW Triggered Actions

A `FOR EACH ROW` triggered action executes once for each row that the triggering statement effects. For example, if the triggering statement is has the following syntax:

```
UPDATE items SET quantity = quantity * 2 WHERE manu_code = 'KAR'
```

A `FOR EACH ROW` triggered action executes once for each row in the **items** table in which the **manu_code** column has a value of 'KAR'.

If the triggering statement does not process any rows, a `FOR EACH ROW` triggered action does not execute.

Using the REFERENCING Clause

When you create a FOR EACH ROW triggered action, you must usually indicate in the triggered action statements whether you are referring to the value of a column before or after the effect of the triggering statement. For example, imagine that you want to track updates to the **quantity** column of the **items** table. To do this, you create the following table to record the activity:

```
CREATE TABLE log_record
  (item_num    SMALLINT,
   ord_num     INTEGER,
   username    CHARACTER(8),
   update_time DATETIME YEAR TO MINUTE,
   old_qty     SMALLINT,
   new_qty     SMALLINT);
```

To supply values for the **old_qty** and **new_qty** columns in this table, you must be able to refer to the old and new values of **quantity** in the **items** table—that is, the values before and after the effect of the triggering statement. The REFERENCING clause enables you to do this.

The REFERENCING clause lets you create two prefixes that you can combine with a column name, one to reference the old value of the column and one to reference its new value. These prefixes are called *correlation names*. You can create one or both correlation names, depending on your requirements. You indicate which one you are creating with the keywords OLD and NEW. The following REFERENCING clause creates the correlation names **pre_upd** and **post_upd** to refer to the old and new values in a row:

```
REFERENCING OLD AS pre_upd NEW AS post_upd
```

The following triggered action creates a row in **log_record** when **quantity** is updated in a row of the **items** table. The INSERT statement refers to the old values of the **item_num** and **order_num** columns and to both the old and new values of the **quantity** column.

```
FOR EACH ROW(INSERT INTO log_record
  VALUES (pre_upd.item_num, pre_upd.order_num, USER, CURRENT,
          pre_upd.quantity, post_upd.quantity));
```

The correlation names defined in the REFERENCING clause apply to all rows affected by the triggering statement.

If you refer to a column name in the triggering table without using a correlation name, the database server makes no special effort to search for the column in the definition of the triggering table. You must always use a correlation name with a column name in SQL statements within a FOR EACH ROW triggered action, *unless* the statement is valid independent of the triggered action. See CREATE TRIGGER in Chapter 1 of the *Informix Guide to SQL: Syntax*.

Using the WHEN Condition

As an option, you can precede a triggered action with a WHEN clause to make the action dependent on the outcome of a test. The WHEN clause consists of the keyword WHEN followed by the condition statement, given in parentheses. In the CREATE TRIGGER statement, the WHEN clause follows the keywords BEFORE, AFTER, or FOR EACH ROW and precedes the triggered action list.

When a WHEN condition is present, if it evaluates to *true*, the triggered actions execute in the order in which they appear. If the WHEN condition evaluates to *false* or *unknown*, the actions in the triggered action list do not execute. If the trigger specifies FOR EACH ROW, the condition is evaluated for each row also.

In the following trigger example, the triggered action executes only if the condition in the WHEN clause is true — that is, if the post-update unit price is greater than two times the pre-update unit price:

```
CREATE TRIGGER up_price
UPDATE OF unit_price ON stock
REFERENCING OLD AS pre NEW AS post
FOR EACH ROW WHEN(post.unit_price > pre.unit_price * 2)
    (INSERT INTO warn_tab VALUES(pre.stock_num, pre.order_num,
    pre.unit_price, post.unit_price, CURRENT))
```

See CREATE TRIGGER in Chapter 1 of the *Informix Guide to SQL: Syntax* for more information on the WHEN condition.

Using Stored Procedures as Triggered Actions

Probably the most powerful feature of triggers is the ability to call a stored procedure as a triggered action. The EXECUTE PROCEDURE statement, which calls a stored procedure, enables you to pass data from the triggering table to the stored procedure and also to update the triggering table with data

returned by the stored procedure. SPL also lets you define variables, assign data to them, make comparisons, and use procedural statements to accomplish complex tasks within a triggered action.

Passing Data to a Stored Procedure

You can pass data to a stored procedure in the argument list of the EXECUTE PROCEDURE statement. The EXECUTE PROCEDURE statement in the following trigger example passes values from the **quantity** and **total_price** columns of the **items** table to the stored procedure **calc_totpr**:

```
CREATE TRIGGER upd_totpr
UPDATE OF quantity ON items
REFERENCING OLD AS pre_upd NEW AS post_upd
FOR EACH ROW(EXECUTE PROCEDURE calc_totpr(pre_upd.quantity,
      post_upd.quantity, pre_upd.total_price) INTO total_price)
```

Passing data to a stored procedure allows you to use it in the operations that the procedure performs.

Using the SPL

The EXECUTE PROCEDURE statement in the preceding trigger calls the stored procedure that is shown in the following example. The procedure uses the SPL to calculate the change that needs to be made to the **total_price** column when **quantity** is updated in the **items** table. The procedure receives both the old and new values of **quantity** and the old value of **total_price**. It divides the old total price by the old quantity to derive the unit price. It then multiplies the unit price by the new quantity to obtain the new total price.

```
CREATE PROCEDURE calc_totpr(old_qty SMALLINT, new_qty SMALLINT,
      total MONEY(8)) RETURNING MONEY(8);
  DEFINE u_price LIKE items.total_price;
  DEFINE n_total LIKE items.total_price;
  LET u_price = total / old_qty;
  LET n_total = new_qty * u_price;
  RETURN n_total;
END PROCEDURE;
```

In this example, SPL enables the trigger to derive data that is not directly available from the triggering table.

Updating Nontriggering Columns with Data from a Stored Procedure

Within a triggered action, the INTO clause of the EXECUTE PROCEDURE statement lets you update nontriggering columns in the triggering table. The EXECUTE PROCEDURE statement in the following example calls the **calc_totpr** stored procedure that contains an INTO clause, which references the column **total_price**:

```
FOR EACH ROW(EXECUTE PROCEDURE calc_totpr(pre_upd.quantity,  
      post_upd.quantity, pre_upd.total_price) INTO total_price);
```

The value that is updated into **total_price** is returned by the RETURN statement at the conclusion of the stored procedure. The **total_price** column is updated for each row that the triggering statement affects.

Tracing Triggered Actions

If a triggered action does not behave as you expect, place it in a stored procedure and use the SPL TRACE statement to monitor its operation. Before starting the trace, you must direct the output to a file with the SET DEBUG FILE TO statement. Figure 15-2 shows TRACE statements that have been added to the stored procedure **items_pct**. The SET DEBUG FILE TO statement directs the trace output to the file **/usr/mydir/trig.trace**. The TRACE ON statement begins tracing the statements and variables within the procedure.

```
CREATE PROCEDURE items_pct(mac CHAR(3))
DEFINE tp MONEY;
DEFINE mc_tot MONEY;
DEFINE pct DECIMAL;
SET DEBUG FILE TO '/usr/mydir/trig.trace';
TRACE 'begin trace';
TRACE ON
LET tp = (SELECT SUM(total_price) FROM items);
LET mc_tot = (SELECT SUM(total_price) FROM items
              WHERE manu_code = mac);
LET pct = mc_tot / tp;
IF pct > .10 THEN
    RAISE EXCEPTION -745;
END IF
TRACE OFF;
END PROCEDURE;

CREATE TRIGGER items_ins
INSERT ON items
REFERENCING NEW AS post_ins
FOR EACH ROW(EXECUTE PROCEDURE items_pct (post_ins.manu_code));
```

Figure 15-2 *Using a stored procedure to trace a triggered action*

Figure 15-3 shows sample trace output from the **items_pct** procedure as it appears in the file **/usr/mydir/trig.trace**. The output reveals the values of procedure variables, procedure arguments, return values, and error codes.

```

trace expression :begin trace
trace on
expression:
  (select (sum total_price)
   from items)
evaluates to $18280.77 ;
let tp = $18280.77
expression:
  (select (sum total_price)
   from items
   where (= manu_code, mac))
evaluates to $3008.00 ;
let mc_tot = $3008.00
expression: (/ mc_tot, tp)
evaluates to 0.16
let pct = 0.16
expression: (> pct, 0.1)
evaluates to 1
expression: (- 745)
evaluates to -745
raise exception :-745, 0, ''
exception : looking for handler
SQL error = -745 ISAM error = 0 error string = ''
exception : no appropriate handler

```

Figure 15-3 Trace output from the **items.pct** stored procedure

See Chapter 14, “Creating and Using Stored Procedures” for more information on using the TRACE statement to diagnose logic errors in stored procedures.

Generating Error Messages

When a trigger fails because of an SQL statement, the database server returns the SQL error number that applies to the specific cause of the failure.

When the triggered action is a stored procedure, you can generate error messages for other error conditions by using one of two reserved error numbers. The first one is error number -745, which has a generalized and fixed error message. The second one is error number -746, which allows you to supply the message text, up to a maximum of 71 characters.

Applying a Fixed Error Message

You can apply error number -745 to any trigger failure that is not an SQL error. The following fixed message is for this error:

-745 Trigger execution has failed.

You can apply this message with the RAISE EXCEPTION statement in SPL. The following example generates error -745 if **new_qty** is greater than **old_qty** multiplied by 1.50:

```
CREATE PROCEDURE upd_items_p2()
  DEFINE GLOBAL old_qty INT DEFAULT 0;
  DEFINE new_qty INT;
  LET new_qty = (SELECT SUM(quantity) FROM items);
  IF new_qty > old_qty * 1.50 THEN
    RAISE EXCEPTION -745;
  END IF
END PROCEDURE
```

If you are using **DB-Access**, the text of the message for error -745 displays on the bottom of the screen, as seen in Figure 15-4.

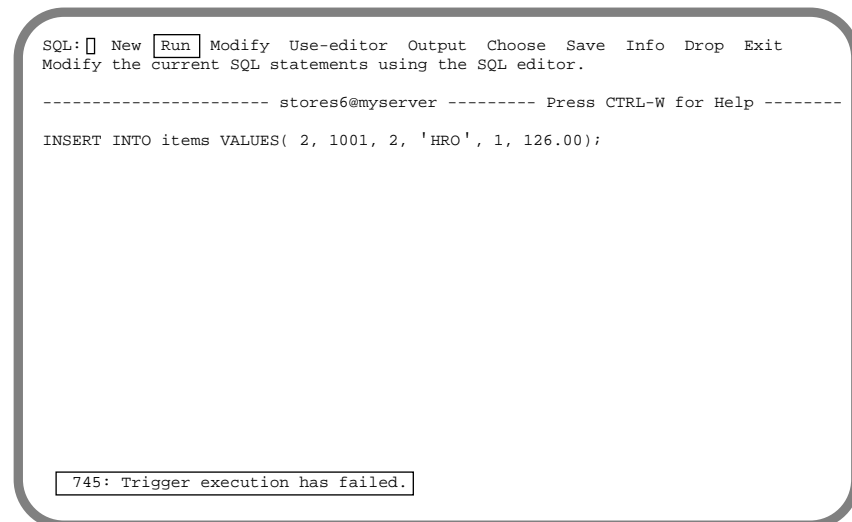


Figure 15-4 Error message -745 with fixed message

If you trigger the erring procedure through an SQL statement in your SQL API, the database server sets the SQL error status variable to -745 and returns it to your program. To display the text of the message, follow the procedure that your Informix application development tool provides for retrieving the text of an SQL error message.

Generating a Variable Error Message

Error number -746 allows you to provide the text of the error message. As with the preceding example, the following one also generates an error if **new_qty** is greater than **old_qty** multiplied by 1.50. However, in this case the error number is -746 and the message text - Too many items for Mfr. - is supplied as the third argument in the RAISE EXCEPTION statement. See the RAISE EXCEPTION statement in Chapter 14, “Creating and Using Stored Procedures” for more information on the syntax and use of this statement.

```
CREATE PROCEDURE upd_items_p2()  
  DEFINE GLOBAL old_qty INT DEFAULT 0;  
  DEFINE new_qty INT;  
  LET new_qty = (SELECT SUM(quantity) FROM items);  
  IF new_qty > old_qty * 1.50 THEN  
    RAISE EXCEPTION -746, 0, 'Too many items for Mfr.';  
  END IF  
END PROCEDURE
```

If you use **DB-Access** to submit the triggering statement, and if **new_qty** is greater than **old_qty**, the result of this stored procedure is as shown in Figure 15-5.

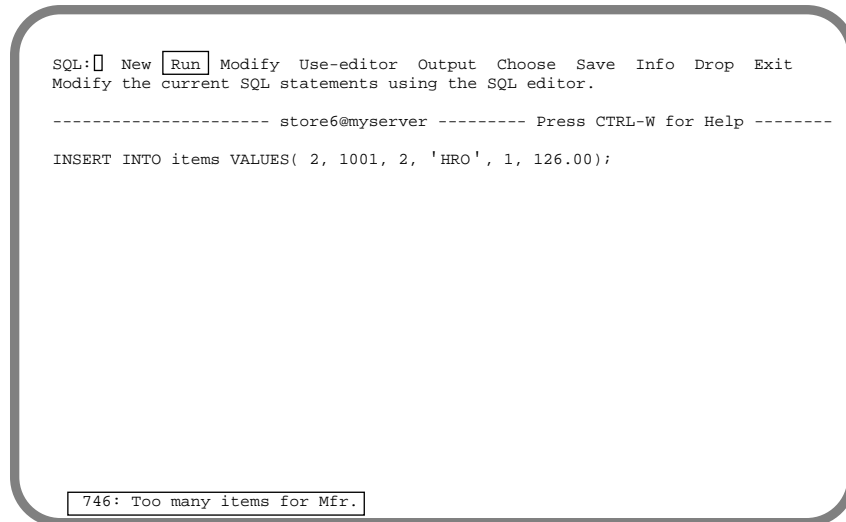


Figure 15-5 Error number -746 with user-specified message text

If you invoke the trigger through an SQL statement in an SQL API, the database server sets **sqlcode** to -746 and returns the message text in the **sqlerrm** field of the SQL Communications Area (SQLCA). See Chapter 5, “Programming with SQL” for your SQL API for a description of the SQLCA.

Summary

In introducing triggers, this chapter explains the following topics:

- The purpose of each component of the CREATE TRIGGER statement
- How to create BEFORE and AFTER triggered actions and how to use them to determine the impact of the triggering statement
- How to create a FOR EACH ROW triggered action and how to use the REFERENCING clause to refer to the values of columns both before and after the action of the triggering statement
- The advantages of using stored procedures as triggered actions
- How to trace triggered actions if they are behaving unexpectedly
- How to generate two types of error messages within a triggered action

Summary

Index

This index covers the Tutorial, Reference, and Syntax manuals. Page numbers that end in T can be found in the Tutorial, those that end in R can be found in the Reference Manual, and those that end in S can be found in the Syntax manual. Special symbols are listed in ASCII order at the end of the index.

A

- ABS function
 - syntax in expression 1-444S
 - use in expression 1-445S
- ABSOLUTE keyword
 - syntax in FETCH 1-194S
 - use in FETCH 1-196S
- Access control. *See* Privilege.
- ACCESS FOR keywords, in INFO statement 1-243S
- Accessing tables 12-16T
- ACOS function
 - syntax in expression 1-458S
 - use in expression 1-460S
- Action clause
 - AFTER 1-117S
 - definition of 1-116S
 - FOR EACH ROW 1-116S
 - subset, syntax 1-121S
 - syntax 1-116S
- Action statements
 - in triggered action clause 1-122S
 - list of 1-122S
 - order of execution 1-122S
- Active set
 - constructing with OPEN 1-264S, 1-265S
 - definition of 2-29T
 - of a cursor 5-24T
 - retrieving data with FETCH 1-194S

ADD CONSTRAINT keywords, syntax
 in ALTER TABLE 1-15S

AFTER
 action 1-117S
 keyword 1-117S

Aggregate function
 ALL keyword, syntax 1-462S
 and GROUP BY clause 3-5T
 AVG function, syntax 1-462S
 description of 2-53T
 DISTINCT keyword, syntax 1-462S
 in ESQL 1-467S, 5-14T
 in EXISTS subquery 1-414S
 in expressions 1-314S
 in SELECT 1-315S
 in SPL expressions 14-21T
 in subquery 3-36T
 MAX function, syntax 1-462S
 MIN function, syntax 1-462S
 null value signalled 5-12T
 restrictions in modifiable view 11-24T
 restrictions with GROUP BY 1-334S
 SUM function, syntax 1-462S
 summary 1-465S

Algebraic functions
 ABS function 1-445S
 MOD function 1-446S
 POW function 1-446S
 ROOT function 1-446S
 ROUND function 1-446S
 SQRT function 1-447S
 syntax 1-444S
 TRUNC function 1-447S

Alias
 for a table in SELECT 1-323S
 for table name 2-79T
 to assign column names in temporary
 table 3-12T
 use in ORDER BY clause 1-338S
 use with GROUP BY clause 1-334S
 with self-join 3-11T
 See also Synonym.

ALL keyword
 beginning a subquery 1-330S, 3-33T
 DISCONNECT statement 1-169S
 syntax
 in expression 1-462S
 in GRANT 1-235S
 in REVOKE 1-301S
 in SELECT 1-312S
 with UNION operator 1-310S

 use
 in Condition subquery 1-415S
 in expression 1-463S
 in GRANT 1-236S
 in SELECT 1-313S
 with UNION operator 1-344S

ALLOCATE DESCRIPTOR statement
 syntax 1-9S
 with concatenation operator 1-432S

Allocating memory with the ALLOCATE
 DESCRIPTOR statement 1-9S

ALTER INDEX statement
 creating clustered index 1-12S, 10-25T
 dropping clustered index 1-13S
 locks table 7-7T
 syntax 1-12S
 See also Index Name segment.

ALTER keyword
 syntax
 in GRANT 1-231S
 in REVOKE 1-301S

 use
 in GRANT 1-236S
 in REVOKE 1-302S

Alter privilege 1-236S, 11-8T

ALTER TABLE statement
 ADD clause 1-17S
 ADD CONSTRAINT clause 1-30S
 adding a column 1-17S
 adding a column constraint 1-29S
 and NCHAR column 1-19R
 and NVARCHAR column 1-19R
 cascading deletes 1-22S
 changing column data type 1-27S,
 9-19T
 changing table lock mode 1-33S
 CHECK clause 1-25S
 DEFAULT clause 1-18S
 DROP clause 1-26S
 DROP CONSTRAINT clause 1-32S
 dropping a column 1-26S
 dropping a column constraint 1-32S
 LOCK MODE clause 1-33S
 MODIFY NEXT SIZE clause 2-9R,
 1-33S
 NEXT SIZE clause 10-8T
 ON DELETE CASCADE keyword
 1-21S
 PAGE keyword 1-33S
 privilege for 1-231S, 11-10T
 reclustering a table 1-13S

REFERENCES clause 1-21S
 ROW keyword 1-33S
 rules for primary key constraints 1-31S
 rules for unique constraints 1-31S
 American National Standards Institute.
 See ANSI.
 AND keyword
 syntax in Condition segment 1-404S
 use
 in Condition segment 1-417S
 with BETWEEN keyword 1-327S
 AND logical operator 1-417S, 2-36T
 ANSI 1-15T
 ANSI compliance
 determining 1-10R
 table naming 1-296S
 ANSI-compliance
 -ansi flag 4-15R, Intro-7S, 1-78S, 1-85S, 1-137S
 described 1-9R
 list of SQL statements 1-7S
 reserved words 1-470S
 updating rows 1-385S
 ANSI-compliant database
 buffered logging restricted in 9-23T
 create with START DATABASE 1-376S
 description of 1-59S, 1-15T
 designating 1-10R
 effect on
 cursor behavior 1-13R
 decimal data type 1-13R
 default isolation level 1-13R
 escape characters 1-13R
 object privileges 1-12R
 owner-naming 1-12R
 SQLCODE 1-14R
 transaction logging 1-12R
 transactions 1-11R
 FOR UPDATE not required in 1-148S, 6-16T
 index naming 1-419S, 1-484S, 1-505S
 owner-naming 1-12R
 privileges 1-12R
 procedure naming 1-495S
 reason for creating 1-9R
 Repeatable Read isolation standard in 7-13T
 signalled in SQLAWARN 5-12T
 SQL statements allowed 1-14R
 table privileges 1-85S, 11-7T
 using with INFORMIX-SE 1-61S
 with BEGIN WORK 1-36S
 ANY keyword
 beginning a subquery 1-330S, 3-34T
 in WHENEVER 1-398S
 use in Condition subquery 1-415S
 Application
 common features 1-18T
 description of 1-16T
 design of order-entry 4-23T
 handling errors 5-16T
 performance analysis of 13-5T
 report generator 1-17T
 screen forms 1-17T
 Archives
 with INFORMIX-SE 12-19T
 Archiving
 description of 1-10T, 4-25T
 INFORMIX-OnLine Dynamic Server methods 4-26T
 transaction log 4-26T
 ARC_DEFAULT environment variable 4-14R
 ARC_KEYPAD environment variable 4-14R
 Arithmetic functions. *See* Algebraic functions.
 Arithmetic operator, in expression 1-431S, 2-46T
 Array, moving rows into with FETCH 1-199S
 AS keyword
 in SELECT 1-312S
 syntax
 in CREATE VIEW 1-136S
 in GRANT 1-231S
 use
 in CREATE VIEW 1-137S
 in GRANT 1-239S
 with display labels 1-316S
 with table aliases 1-325S
 ASC keyword
 syntax
 in CREATE INDEX 1-63S
 in SELECT 1-337S
 use
 in CREATE INDEX 1-66S
 in SELECT 1-338S

Ascending order in SELECT 2-14T
 ASCII collating order 1-15R, 1-501S
 ASIN function
 syntax in expression 1-458S
 use in expression 1-460S
 Asterisk
 wild card character in SELECT 2-12T
 Asterisk (*)
 arithmetic operator 1-431S
 use in SELECT 1-312S
 At (@) sign, in database name 1-421S
 ATAN function
 syntax in expression 1-458S
 use in expression 1-460S
 ATAN2 function
 syntax in expression 1-458S
 use in expression 1-461S
 Attribute
 identifying 8-14T
 important qualities of 8-15T
 nondecomposable 8-15T
 Audit trail
 applying with RECOVER TABLE
 1-292S
 dropping with DROP AUDIT 1-171S
 manipulating audit trail file 1-293S
 no clustered index 1-64S
 starting with CREATE AUDIT 1-55S
 Automatic type conversion. *See* Data
 type conversion.
 AVG function
 as aggregate function 2-53T
 syntax in expression 1-462S
 use in expression 1-464S

B

Bachman, C.R. 8-17T
 Backslash (\)
 as escape character with LIKE 1-410S
 as escape character with MATCHES
 1-411S
 Backup. *See* Archive.
 BEFORE keyword 1-116S
 BEGIN WORK statement
 locking in a transaction 1-35S
 specifies start of a transaction 4-24T
 syntax 1-35S
 BETWEEN keyword
 syntax in Condition segment 1-405S
 use
 in Condition segment 1-408S
 in SELECT 1-327S
 BETWEEN keyword, used to test for
 equality in WHERE clause 2-29T
 BETWEEN operator 2-32T
 Binary Large Object (BLOB)
 choosing location for 10-18T
 disk storage for 10-4T, 10-6T
 effect of isolation on retrieval 1-368S
 estimating disk space for 10-17T
 in a LOAD statement 1-257S
 in an UNLOAD statement 1-379S
 See also BYTE data type.
 See also TEXT data type.
 blobspace 10-4T, 10-18T
 BLOB. *See* Binary Large Object.
 Boolean expression
 and logical operator 2-36T
 in Condition segment 1-404S
 Bourne shell
 how to set environment variables
 4-5R
 .profile file 4-4R
 BUFFERED keyword, syntax in SET LOG
 1-372S
 BUFFERED LOG keywords
 syntax in CREATE DATABASE 1-57S
 use in CREATE DATABASE 1-59S
 Buffered logging 1-57S, 9-22T
 Building your data model 8-3T to 8-33T
 BYTE data type
 choosing location for 10-18T
 considerations for UNLOAD
 statement 1-379S
 description of 3-5R, 9-19T
 disk storage for 10-4T
 estimating disk space for 10-17T
 inserting data 3-5R
 requirements for LOAD statement
 1-257S
 restrictions
 in Boolean expression 3-5R
 with GROUP BY 3-5R
 with LIKE or MATCHES 3-5R
 with ORDER BY 3-5R
 restrictions with GROUP BY 3-7T
 restrictions with LIKE or MATCHES
 2-37T

-
- restrictions with relational expression 2-29T
 - selecting a BYTE column 3-5R
 - syntax 1-425S
 - with stored procedures 2-9S, 2-13S
- BYTE value, displaying 2-11T
- ## C
- C shell
 - how to set environment variables 4-5R
 - .cshrc file 4-4R
 - .login file 4-4R
 - Calculated expression
 - restrictions with GROUP BY 1-334S
 - See also* Expression segment.
 - CALL keyword, in the WHENEVER statement 1-398S, 1-401S
 - CALL statement
 - assigning values with 14-21T
 - executing a procedure 14-9T
 - syntax 2-3S
 - call_type table in stores6 database, columns in A-5R
 - Candidate key
 - defined 8-23T
 - Cardinality 8-9T
 - Cardinality in relationship 8-13T
 - Caret (^) wildcard in Condition segment 1-411S
 - Cartesian product
 - basis of any join 2-71T
 - description of 2-69T
 - Cascading deletes
 - defined 1-22S, 1-97S, 4-21T
 - locking associated with 1-24S, 1-98S, 4-21T
 - logging 1-24S, 1-98S, 4-21T
 - restriction 1-24S, 1-98S, 4-22T
 - syntax 1-21S, 1-94S
 - Cascading triggers
 - and triggering table 1-127S, 1-131S
 - description of 1-130S
 - maximum number of 1-130S
 - scope of correlation names 1-125S
 - triggered actions 1-118S
 - Catalog. *See* System catalog.
 - Chaining synonyms 12-18T
 - CHAR data type 9-15T
 - changing data types 3-23R
 - description of 3-6R
 - in INSERT 1-498S
 - in relational expressions 2-29T
 - replacing with TEXT 10-27T
 - replacing with VARCHAR 10-26T
 - subscripting 2-44T
 - substrings of 2-27T
 - syntax 1-425S
 - truncation signalled 5-12T
 - using as default value 1-19S, 1-89S
 - versus NCHAR data type 1-19R
 - CHARACTER data type. *See* CHAR data type.
 - Character mapping files 5-5R
 - Character string
 - as DATE values 3-29R
 - as DATETIME values 3-10R, 3-29R
 - as INTERVAL values 3-15R
 - processing with NLS 1-15R
 - See also* Quoted String segment.
 - Character-position form of FILE and INSERT statements 5-28R
 - Check constraint
 - adding with ALTER TABLE 1-25S
 - definition of 1-98S, 4-19T
 - specifying at column level 1-98S
 - specifying at table level 1-98S
 - CHECK keyword
 - use in ALTER TABLE 1-25S
 - use in CREATE TABLE 1-98S
 - CHECK TABLE statement, syntax and use 1-37S
 - Checking contents of environment configuration file 5-4R
 - Checking for corrupted tables 1-37S
 - chkenv utility
 - description of 5-4R
 - error message for 5-4R
 - Chunk
 - description of 10-4T
 - mirrored 10-5T
 - Client/server environment 1-133S
 - CLOSE DATABASE statement
 - effect on database locks 7-7T
 - prerequisites to close 1-41S
 - syntax 1-41S
 - CLOSE statement

- closing
 - a select cursor 1-38S
- closing an insert cursor 1-39S
- cursors affected by transaction end 1-40S
- syntax 1-38S
- with concatenation operator 1-432S
- CLUSTER keyword
 - syntax
 - in ALTER INDEX 1-12S
 - in CREATE INDEX 1-63S
 - use
 - in ALTER INDEX 1-12S
 - in CREATE INDEX 1-64S
- Clustered index
 - creating with CREATE INDEX 1-64S
 - description of 10-25T
 - with ALTER INDEX 1-12S
 - with audit trails 1-64S
- COBOL 5-6T
- Codd, E. F. 1-11T, 8-4T, 8-32T
- Code set
 - for crctmap text file 5-6R
 - mapping non-standard to standard 5-5R
 - mapping with crctmap utility 5-5R
- Collation
 - COLLCHAR environment variable 4-45R
 - LC_COLLATE environment variable 4-50R
 - simultaneous, and performance 1-20R
 - with NLS activated 1-15R
- Collation order and NLS 2-25T
- COLLCHAR environment variable 1-16R, 4-45R
- Colon (:)
 - as delimiter in DATETIME 3-9R
 - as delimiter in INTERVAL 3-15R
- Color, setting INFORMIXTERM for 4-38R
- Column
 - changing data type 3-23R
 - creating with NLS 1-19R
 - defined 2-5T
 - defining 8-20T
 - defining as foreign key 1-93S
 - defining as primary key 1-93S
 - description of 1-12T
 - displaying information for 1-242S
 - in relational model 1-12T, 8-20T
 - in stores6 database A-2R to A-6R
 - inserting into 1-246S
 - label on 3-48T
 - modifying with ALTER TABLE 1-27S
 - naming conventions 1-87S, 1-102S, 1-294S
 - naming, allowable characters 1-469S
 - naming, in ALTER TABLE 1-17S
 - naming, in CREATE TABLE 1-87S
 - number allowed when defining constraint 1-86S
 - putting a constraint on 1-86S
 - referenced and referencing 1-22S, 1-94S
 - renaming 1-294S
 - specifying check constraint for 1-98S
 - specifying with CREATE TABLE 1-87S
 - virtual 1-137S
 - See also* Constraint.
 - See also* Data Type segment.
- Column expression
 - in SELECT 1-314S
 - syntax 1-433S
 - See also* Expression segment.
- Column filter. *See* Filter expression.
- Column name
 - allowable characters 1-469S
 - in UPDATE clause 1-114S
 - naming in ALTER TABLE 1-17S
 - using functions as names 1-472S
 - using keywords as names 1-473S
 - when qualified 1-124S
- Column number 2-24T
 - effect on triggers 1-115S
- Column value
 - in triggered action 1-125S
 - qualified vs. unqualified 1-126S
 - when unqualified 1-125S
- Column-level privilege 1-236S, 11-10T
- COLUMNS FOR keywords, in INFO statement 1-242S
- Command file, dbload 5-23R
- Command script, creating database 9-26T
- Commit, two-phase 12-20T
- COMMIT WORK statement
 - closes cursors 7-18T
 - releases locks 7-8T, 7-18T

sets SQLCODE 6-5T
 syntax 1-43S
 Committed Read isolation level 1-367S, 7-10T
 COMMITTED READ keywords, syntax in SET ISOLATION 1-366S
 Comparison condition
 description of 2-29T
 syntax and use 1-405S
 See also Boolean expression.
 Compiler
 setting environment variable for C 4-31R
 setting environment variable for COBOL 4-32R, 4-33R
 specifying storage mode for COBOL 4-33R
 Complex
 relationship 8-28T
 Complex condition. *See* Condition segment.
 Complex query
 example of 1-364S
 Composite column list, multiple-column restrictions 1-30S, 1-31S
 Composite index
 column limit 1-65S
 creating with CREATE INDEX 1-63S
 definition of 1-65S
 order of columns 13-31T
 use 13-31T
 Composite key 8-23T
 Compound assignment 2-28S
 Compound query 3-43T
 Computer network 12-4T
 Concatenation operator (||) 1-432S
 Concurrency
 Committed Read isolation 1-367S, 7-10T
 Cursor Stability isolation 1-367S, 7-11T
 database lock 7-6T
 deadlock 7-14T
 defining with SET ISOLATION 1-366S
 description of 4-27T, 7-3T
 Dirty Read isolation 1-367S, 7-10T
 effect on performance 7-3T
 isolation level 7-9T
 kinds of locks 7-6T
 lock duration 7-8T
 lock scope 7-6T
 maximizing 10-32T, 10-35T
 Repeatable Read isolation 1-367S, 7-12T
 SERIAL values 9-7T
 table lock 7-7T
 Condition segment
 ALL, ANY, SOME subquery 1-415S
 boolean expressions 1-405S
 comparison condition 1-405S
 description of 1-404S
 join conditions 1-331S
 null values 1-405S
 relational operators in 1-407S
 subquery in SELECT 1-413S
 syntax 1-404S
 use of functions in 1-405S
 wildcards in searches 1-410S
 with BETWEEN keyword 1-408S
 with ESCAPE keyword 1-411S
 with EXISTS keyword 1-414S
 with IN keyword 1-408S
 with IS keyword 1-409S
 with MATCHES keyword 1-409S
 with NOT keyword 1-410S
 Configuring a database server 12-15T
 CONNECT keyword
 in GRANT 1-232S
 in REVOKE 1-303S
 Connect privilege 1-232S, 1-303S, 11-6T
 CONNECT statement
 and INFORMIXSERVER
 environment variable 4-37R, 1-46S
 connection context 1-45S
 connection identifiers 1-45S
 database environment 1-49S
 DEFAULT option 1-46S
 implicit connections 1-46S
 opening a database 12-16T
 syntax 1-44S
 use 1-44S
 USER clause 1-52S
 WITH CONCURRENT
 TRANSACTION option 1-47S
 CONNECT TO statement
 with concatenation operator 1-432S
 Connecting to data 12-12T

Connection
 context 1-45S
 dormant 1-169S
 identifiers 1-45S
 setting the INFORMIXCONRETRY environment variable 4-34R
 setting the INFORMIXCONTIME environment variable 4-35R
 Connectivity in relationship 8-8T, 8-11T, 8-17T
 Constant expression
 in SELECT 1-314S
 inserting with PUT 1-286S
 restrictions with GROUP BY 1-334S
 syntax 1-436S
 See also Expression segment.
 See also Literal Number.
 Constraint
 adding with ALTER TABLE 1-29S, 1-86S
 cardinality 8-9T
 checking 1-131S
 defining domains 9-3T
 definition of 1-85S
 dropping with ALTER TABLE 1-32S, 1-86S
 enforcing 1-86S
 modifying a column that has constraints 1-27S
 number of columns allowed 1-86S, 1-92S
 optimizer uses 13-9T
 privileges needed to create 1-31S
 rules for unique constraints 1-30S
 setting checking mode 1-349S
 specifying at table level 1-92S
 with DROP INDEX 1-174S
 See also Primary key constraint.
 See also Referential constraint.
 See also Unique constraint.
 CONSTRAINT keyword
 in ALTER TABLE 1-29S
 in CREATE TABLE 1-91S
 Contention
 for bottleneck tables 10-35T
 for disk access arms 10-7T
 reducing 10-32T, 10-35T
 CONTINUE keyword, in the WHENEVER statement 1-398S, 1-402S
 CONTINUE statement
 exiting a loop 14-23T
 syntax 2-6S
 Conventions
 example code Intro-8R, Intro-10S, 5T
 for naming tables 1-84S
 syntax Intro-5S
 typographical Intro-5R, Intro-5S, 5T
 Converting data types 3-23R
 Coordinated deletes 6-6T
 Correlated subquery
 definition of 1-413S, 3-32T
 restriction with cascading deletes 4-22T
 Correlation name
 and stored procedures 1-125S
 in COUNT DISTINCT clause 1-125S
 in DELETE REFERENCING clause 1-119S
 in GROUP BY clause 1-125S
 in INSERT REFERENCING clause 1-118S
 in SET clause 1-125S
 in stored procedure 1-128S
 in UPDATE REFERENCING clause 1-120S
 new 1-120S
 old 1-120S
 rules for 1-124S
 scope of 1-125S
 table of values 1-126S
 using 1-124S
 when to use 1-125S
 COS function
 syntax in expression 1-458S
 use in expression 1-459S
 COUNT DISTINCT clause 1-125S
 COUNT field
 getting contents with GET DESCRIPTOR 1-210S
 setting value for WHERE clause 1-354S
 use in GET DESCRIPTOR 1-212S
 COUNT function
 and GROUP BY 3-6T
 as aggregate function 2-53T
 count rows to delete 4-5T
 use in a subquery 4-6T
 use in expression 1-463S, 1-465S
 with DISTINCT 2-54T

COUNT keyword, use in SET
 DESCRIPTOR 1-355S
 CREATE AUDIT statement
 need for archive 1-55S
 starts audit trail 1-55S
 syntax 1-55S
 CREATE DATABASE statement
 and dbspace 10-5T
 ANSI compliance 1-59S
 in command script 9-26T
 logging with OnLine 1-59S
 sets shared lock 7-6T
 SQLAWARN after 5-12T
 syntax 1-57S
 using with
 CREATE SCHEMA 1-77S
 INFORMIX-SE 1-60S, 9-23T
 OnLine 9-21T
 PREPARE 1-58S
 CREATE INDEX statement
 composite indexes 1-65S
 implicit table locks 1-63S
 locks table 7-7T
 syntax 1-62S
 using
 with CREATE SCHEMA 1-77S
 using with
 ASC keyword 1-66S
 CLUSTER keyword 1-64S
 DESC keyword 1-66S
 UNIQUE keyword 1-64S
 See also Index Name segment.
 CREATE PROCEDURE FROM
 statement
 in embedded languages 14-5T
 syntax and use 1-76S
 CREATE PROCEDURE statement
 inside a CREATE PROCEDURE
 FROM 14-5T
 syntax 1-68S
 use of dbschema 5-34R
 using 14-5T
 CREATE SCHEMA statement
 defining a trigger 1-111S
 syntax 1-77S
 with CREATE sequences 1-78S
 with GRANT 1-78S
 CREATE SYNONYM statement
 ANSI-compliant naming 1-80S
 chaining synonyms 1-83S
 synonym for a table 1-80S
 synonym for a view 1-80S
 syntax 1-80S
 use of dbschema 5-34R
 with CREATE SCHEMA 1-77S
 CREATE TABLE
 use of dbschema 5-34R
 CREATE TABLE statement
 and COLLCHAR environment
 variable 4-46R
 and NCHAR column 1-19R
 and NVARCHAR column 1-19R
 cascading deletes 1-97S
 CHECK clause 1-98S
 creating temporary table 1-100S
 DEFAULT clause 1-88S
 defining constraints
 at column level 1-91S
 at table level 1-92S
 description of 9-24T
 EXTENT SIZE clause 10-8T
 in command script 9-26T
 IN dbspace clause 1-105S
 locating BLOB column 10-18T
 LOCK MODE clause 1-108S
 naming conventions 1-84S
 NEXT SIZE clause 10-8T
 ON DELETE CASCADE keyword
 1-94S
 REFERENCES clause 1-94S
 rules for primary keys 1-94S
 rules for referential constraints 1-94S
 rules for unique constraints 1-94S
 sets initial SERIAL value 9-8T
 setting columns NOT NULL 1-20S,
 1-90S
 specifying extent size 1-107S
 specifying table columns 1-87S
 storing database tables 1-105S
 syntax 1-84S
 with BLOB data types 1-91S
 with CREATE SCHEMA 1-77S
 CREATE TRIGGER statement
 in ESQL/C 1-111S
 in ESQL/COBOL 1-111S
 privilege to use 1-111S
 purpose 1-110S
 syntax 1-110S
 triggered action clause 1-121S
 use 1-111S
 CREATE VIEW statement

-
- column data types 1-137S
 - privileges 1-137S
 - restrictions on 11-22T
 - syntax 1-136S
 - use of dbschema 5-34R
 - using 11-20T
 - virtual column 1-137S
 - WITH CHECK OPTION 1-138S
 - with CREATE SCHEMA 1-77S
 - with SELECT * notation 1-136S
 - Creating
 - a database from ASCII files 5-14R
 - a dbload command file 5-23R
 - crtcmmap utility
 - creating mapping files 5-5R
 - description of 5-5R
 - error messages for 5-7R
 - formats for mapping files 5-6R
 - Currency, representing with NLS 1-15R
 - Current database
 - specifying with DATABASE 1-140S
 - CURRENT function
 - 4GL example 13-6T
 - comparing column values 2-56T
 - syntax
 - in Condition segment 1-405S
 - in expression 1-436S
 - in INSERT 1-250S
 - use
 - in ALTER TABLE 1-18S
 - in CREATE TABLE 1-88S
 - in expression 1-440S
 - in INSERT 1-252S
 - in WHERE condition 1-441S
 - input for DAY function 1-441S
 - CURRENT keyword
 - DISCONNECT statement 1-168S
 - syntax in FETCH 1-194S
 - use in FETCH 1-196S
 - CURRENT OF keywords
 - syntax
 - in DELETE 1-159S
 - in UPDATE 1-383S
 - use
 - in DELETE 1-161S
 - in UPDATE 1-390S
 - Cursor
 - activating with OPEN 1-263S
 - active set of 5-24T
 - affected by transaction end 1-40S
 - associating with prepared statements 1-154S
 - characteristics 1-149S
 - closing 1-38S, 7-18T
 - closing with ROLLBACK WORK 1-307S
 - declaring 1-145S, 5-20T
 - definition of types 1-148S
 - for insert 6-8T
 - for update 6-15T, 7-9T
 - hold 7-18T
 - manipulation statements 1-6S
 - opening 1-264S, 1-265S, 5-20T, 5-24T
 - retrieving values with FETCH 1-194S, 5-21T
 - scroll 1-149S, 5-23T
 - sequential 1-149S, 5-23T, 5-25T
 - statement, as trigger event 1-112S
 - using with transactions 1-155S
 - with
 - DELETE 1-159S
 - INTO keyword in SELECT 1-319S
 - prepared statements 1-148S, 5-32T
 - WITH HOLD 7-18T, 10-34T
 - See also* CLOSE statement.
 - See also* Hold cursor.
 - See also* Insert cursor.
 - See also* OPEN statement.
 - See also* Scroll cursor.
 - See also* Select cursor.
 - See also* Sequential cursor.
 - See also* Update cursor.
 - Cursor Stability isolation level 1-367S, 7-11T
 - CURSOR STABILITY keywords, syntax in SET ISOLATION 1-366S
 - Cursory procedure 2-21S
 - customer table in stores6 database, columns in A-2R
 - cust_calls table in stores6 database, columns in A-5R
 - Cyclic query 4-22T
- ## D
- Daemon 12-14T
 - Data
 - connecting to 12-12T
 - containing foreign characters 1-19R
 - inserting with the LOAD statement 1-255S

- integrity 12-20T
- Data access statements 1-6S
- Data definition statements 1-5S, 5-34T
- Data distributions
 - confidence 1-397S
 - creating on filtered columns 13-9T, 13-29T
 - on temporary tables 1-397S
 - RESOLUTION 1-396S, 1-397S
 - use with optimizer 13-9T
- DATA field
 - setting with SET DESCRIPTOR 1-357S
- Data integrity 4-22T to 4-25T, 12-20T
 - statements 1-6S
- Data manipulation statements 1-6S
- Data model
 - attribute 8-14T
 - building 8-3T to 8-33T
 - defining relationships 8-8T
 - denormalizing 10-26T to 10-32T
 - description of 1-3T, 8-3T
 - entity-relationship 8-4T
 - many-to-many relationship 8-11T
 - one-to-many relationship 8-11T
 - one-to-one relationship 8-11T
 - See also* Relational model.
 - telephone-directory example 8-6T
- Data protection
 - with OnLine 12-19T
- Data replication 12-19T
- Data type
 - automatic conversions 5-15T
 - BYTE 3-5R, 9-19T
 - changing with ALTER TABLE 1-28S
 - CHAR 3-6R, 9-15T
 - CHARACTER 3-7R
 - character data 9-15T
 - CHAR, mapping to NCHAR 4-46R
 - choosing 9-19T
 - chronological 9-12T
 - considerations for INSERT 1-251S, 1-498S
 - conversion 3-23R, 4-8T
 - DATE 3-7R, 9-12T
 - DATETIME 3-7R, 9-12T
 - DEC 3-10R
 - DECIMAL 3-10R, 9-10T
 - DOUBLE PRECISION 3-12R
 - fixed-point 9-10T
 - FLOAT 3-12R
 - floating-point 3-12R, 9-9T
 - in SPL variables 14-17T
 - INT 3-13R
 - INTEGER 3-13R, 9-7T
 - INTERVAL 3-13R, 9-14T
 - MONEY 3-16R, 9-10T
 - NCHAR 3-17R
 - NCHAR, mapping to CHAR 4-46R
 - NUMERIC 3-18R
 - numeric 9-7T
 - NVARCHAR 3-18R
 - REAL 3-19R, 9-9T
 - requirements for referential constraints 1-22S, 1-96S
 - segment 1-424S
 - SERIAL 3-19R, 9-7T
 - SMALLFLOAT 3-20R, 9-9T
 - SMALLINT 3-20R
 - specifying with CREATE VIEW 1-137S
 - summary table 3-4R
 - syntax 1-425S
 - TEXT 3-21R, 9-17T
 - VARCHAR 3-22R, 9-15T
 - See also* Data Type segment.
- Data types
 - creating NCHAR columns 1-19R
 - creating NVARCHAR columns 1-19R
 - OnLine specific 1-4R
- Database
 - ANSI-compliant 1-15T
 - application 1-16T
 - archiving 1-10T
 - closing with CLOSE DATABASE 1-41S
 - concurrent use 1-8T
 - creating ANSI-compliant 1-376S
 - creating in NLS mode 1-20R
 - creating with CREATE DATABASE 1-57S
 - data types 3-4R
 - default isolation levels 1-368S
 - defined 1-3T, 1-11T
 - dropping 1-172S
 - implicit connection 1-168S
 - lock 1-142S
 - management of 1-9T
 - map of
 - stores6 A-6R
 - system catalog tables 2-33R
 - mission-critical 1-10T
 - naming conventions 1-422S

- naming unique to engine 9-21T
- naming with variable 1-423S
- NLS 1-16T
- NLS versus non-NLS 1-20R
- NLS, accessing 1-20R
- opening in exclusive mode 1-142S
- optimizing queries 1-394S
- populating new tables 9-27T
- relation to dbspace 10-5T
- relational, defined 1-11T
- remote 1-422S
- restoring 1-308S
- server 1-8T, 1-16T
- server, definition of 1-16T
- stopping logging on 1-376S
- stores6 Intro-9R, Intro-12S, 7T
- stores6 description of A-1R
- table names 12-16T
- See also* Database Name segment.
- Database Administrator (DBA) 1-233S, 11-7T
- Database application. *See* Application.
- Database lock 7-6T
- Database management system 12-4T
- Database Name segment
 - database outside DBPATH 1-423S
 - for remote database 1-422S
 - naming conventions 1-421S
 - naming with variable 1-423S
 - syntax 1-421S
 - using quotes, slashes 1-423S
- Database server
 - choosing OnLine or SE 1-3R
 - configuration 12-15T
 - definition of 1-16T
 - effect of server type on
 - available data types 1-4R
 - isolation level 1-6R
 - locking 1-5R
 - rolling back transactions 1-4R
 - SQL statements supported 1-7R
 - system catalog tables 1-7R
 - transaction logging 1-5R
 - local 12-5T
 - NLS versus non-NLS 1-20R
 - remote 12-8T
 - specifying default for connection 4-36R
- DATABASE statement
 - determining database type 1-140S
 - exclusive mode 1-142S, 7-6T
 - for database outside DBPATH 1-141S
 - locking 7-6T
 - specifying current database 1-140S
 - SQLAWARN after 1-141S, 5-12T
 - syntax 1-140S
 - using with program variables 1-141S
 - See also* Database Name segment.
- Database-level privilege
 - description of 1-232S, 4-15T
 - granting 1-232S
 - passing grant ability 1-237S
 - revoking 1-303S
 - See also* Privilege.
- DATE data type
 - converting to DATETIME 3-25R
 - description of 3-7R, 9-12T
 - functions in 1-454S, 2-56T
 - in ORDER BY sequence 2-14T
 - range of operations 3-25R
 - representing DATE values 3-29R
 - syntax 1-425S
 - using with DATETIME and INTERVAL values 3-28R
- DATE function
 - as time function 2-56T
 - syntax in expression 1-454S
 - use in expression 1-455S, 2-61T
- DATE value
 - setting DBDATE environment variable 4-16R
 - specifying European format with DBDATE 4-17R
- DATETIME data type
 - 4GL example 13-6T
 - adding or subtracting INTERVAL values 3-27R
 - as quoted string 1-498S
 - character string values 3-10R
 - converting to DATE 3-25R
 - description of 9-12T
 - displaying format 2-61T, 9-16T
 - field qualifiers 3-8R, 1-428S
 - functions on 2-56T
 - in
 - expression 1-441S
 - INSERT 1-498S
 - ORDER BY sequence 2-14T
 - relational expressions 2-29T
 - multiplying values 3-26R
 - precision and size 3-8R, 9-13T
 - range of expressions 3-26R

range of operations with DATE and INTERVAL 3-25R
 representing DATETIME values 3-29R
 syntax 1-425S, 1-488S
 using the DBTIME environment variable 4-27R
 with EXTEND function 3-26R, 3-28R
See also Literal DATETIME.

DATETIME Field Qualifier segment 1-428S

DATETIME formats, using the DBTIME environment variable 4-27R

Date, representing with NLS 1-15R

DAY function
 as time function 2-57T
 syntax in expression 1-454S
 use
 as time function 2-56T
 in expression 1-455S

DAY keyword
 syntax
 in DATETIME data type 1-428S
 in INTERVAL data type 1-485S
 use
 as DATETIME field qualifier 3-8R, 1-488S
 as INTERVAL field qualifier 3-14R, 1-491S

DBA keyword
 in GRANT 1-233S
 in REVOKE 1-303S

DB-Access
 creating database with 5-34T, 9-26T
 UNLOAD statement 9-28T

DBANSIWARN environment variable 4-15R, 1-78S, 1-85S, 1-137S, 5-10T

DBAPICODE environment variable 1-16R, 4-47R
 relation to crctmap utility 5-5R

DBA-privileged procedure 14-13T

DBA. *See* Database Administrator.

DBDATE environment variable 4-16R, 4-8T, 9-13T

DBDELIMITER environment variable 4-18R, 1-258S

DBEDIT environment variable 4-18R

dbexport utility
 description of 5-8R
 destination options 5-10R
 Interrupt key 5-9R
 specifying field delimiter with DBDELIMITER 4-18R
 unloading a database 5-8R
 using with NLS 5-12R

dbimport utility
 create options 5-16R
 creating a database 5-14R
 description of 5-13R
 input file location options 5-15R
 Interrupt key 5-14R
 using with NLS 5-13R, 5-18R

DBINFO function
 syntax in expression 1-448S
 use in expression 1-448S

DBLANG environment variable 4-19R

dbload utility
 creating a command file 5-23R
 description of 5-19R
 INSERT statement, compared to SQL INSERT statement 5-30R
 Interrupt key 5-21R
 loading data from a command file 5-20R
 loading data into a table 9-28T, 10-12T
 options
 bad-row limits 5-23R
 batch size 5-22R
 command-file syntax check 5-21R
 load start point 5-22R
 specifying field delimiter with DBDELIMITER 4-18R
 writing a command file in character-position form 5-31R
 writing a command file in delimiter form 5-26R

DBMONEY environment variable 4-20R, 9-12T

DBMS. *See* Database management system.

DBNLS environment variable 1-16R, 4-48R

DBPATH environment variable 4-21R, 1-141S, 1-423S, 12-14T

DBPRINT environment variable 4-24R

DBREMOTECMD environment variable 4-25R

Directory, extension,.dbs extension 1-101S

.dbs extension Intro-11R, 1-58S, 1-141S

- dbschema utility
 - create schema for a database 5-34R
 - description of 5-33R
 - options
 - obtaining privilege schema 5-36R
 - obtaining synonym schema 5-35R
 - specifying a table, view, or procedure 5-37R
 - owner conventions 5-35R
 - use of 9-26T
- DBSERVERNAME function
 - returning servername 1-438S
 - use
 - in ALTER TABLE 1-18S
 - in CREATE TABLE 1-88S
 - in expression 1-438S
 - in SELECT 2-63T, 2-65T, 3-20T
- dbspace
 - definition of 10-4T
 - division into extents 10-8T
 - for temporary tables 10-6T
 - mirrored 10-5T
 - multiple access arms in 10-7T
 - on dedicated device 10-6T
 - relation to tbspace 10-7T
 - root 10-5T
 - selecting with CREATE DATABASE 1-57S, 9-22T
- DBSPACETEMP environment variable 4-26R, 1-100S
- DBTEMP environment variable 4-27R
- DBTIME environment variable 4-27R
- DBUPSPACE environment variable 4-30R
- DDL statements, summary 1-5S
- Deadlock detection 1-371S, 7-14T
- DEALLOCATE DESCRIPTOR statement
 - syntax 1-143S
 - with concatenation operator 1-432S
- DEC data type. *See* DECIMAL data type.
- DECIMAL data type
 - changing data types 3-23R
 - description of 3-11R
 - fixed-point 9-10T
 - floating-point 3-11R, 9-10T
 - signalled in SQLAWARN 5-12T
 - syntax 1-425S
 - using as default value 1-19S, 1-89S
- Decimal point (.)
 - as delimiter in DATETIME 3-9R
 - as delimiter in INTERVAL 3-15R
- DECLARE statement
 - cursor characteristics 1-149S
 - cursor types 1-148S
 - cursors with prepared statements 1-154S
 - cursors with transactions 1-155S
 - definition and use
 - hold cursor 1-150S
 - insert cursor 1-149S, 1-157S
 - procedure cursor 1-148S
 - scroll cursor 1-149S
 - select cursor 1-148S
 - sequential cursor 1-149S
 - update cursor 1-148S, 1-152S
 - description of 5-20T
 - FOR INSERT clause 6-8T
 - FOR UPDATE 6-15T
 - insert cursor 1-148S
 - insert cursor with hold 1-157S
 - procedure cursor 1-148S
 - restrictions with SELECT with ORDER BY 1-340S
 - SCROLL keyword 5-23T
 - syntax 1-145S
 - update cursor 1-148S
 - updating specified columns 1-153S
 - use
 - with concatenation operator 1-432S
 - with FOR UPDATE keywords 1-148S
 - WITH HOLD clause 7-19T
 - with SELECT 1-321S
- Default assumptions for your environment 4-6R
- DEFAULT keyword 1-169S
 - in the CONNECT statement 1-46S
- Default value
 - description of 4-19T
 - specifying
 - with ALTER TABLE 1-19S
 - with CREATE TABLE 1-88S
- Deferred checking 1-349S
- DEFERRED keyword, in the SET CONSTRAINTS statement 1-349S
- DEFINE statement
 - in stored procedures 14-17T
 - placement of 2-8S
 - syntax 2-7S

DELETE keyword
 syntax
 in GRANT 1-235S
 in REVOKE 1-301S
 use
 in GRANT 1-236S
 in REVOKE 1-302S
 Delete privilege 1-235S, 11-8T, 11-28T
 DELETE REFERENCING clause
 and FOR EACH ROW section 1-121S
 correlation name 1-119S
 syntax 1-119S
 DELETE statement
 all rows of table 4-4T
 and end of data 6-14T
 applied to view 11-24T
 as triggering statement 1-112S
 cascading 1-160S
 coordinated deletes 6-6T
 count of rows 6-4T
 CURRENT OF clause 1-161S
 description of 4-4T
 embedded 5-6T, 6-3T to 6-8T
 in trigger event 1-111S
 in triggered action 1-122S
 number of rows 5-12T
 preparing 5-30T
 privilege for 1-235S, 11-6T, 11-8T
 syntax 1-159S
 time to update indexes 10-20T
 transactions with 6-5T
 using subquery 4-6T
 WHERE clause restricted 4-6T
 with Condition segment 1-404S
 with cursor 1-152S, 6-7T
 with select..for update 1-340S
 within a transaction 1-159S
 Delete, cascading
 See Cascading deletes
 Delimiter
 for DATETIME values 3-9R
 for INTERVAL values 3-15R
 for LOAD input file 1-258S
 specifying with UNLOAD 1-380S
 Delimiter form of FILE and INSERT
 statements 5-24R
 DELIMITER keyword
 in LOAD 1-258S
 in UNLOAD 1-380S
 Demonstration database
 copying Intro-10R, Intro-13S, 8T
 installation script Intro-9R, Intro-12S, 7T
 map of A-6R
 overview Intro-9R, Intro-12S, 7T
 structure of tables A-2R
 tables in A-2R to A-6R
 See also stores6 database.
 Denormalizing 10-26T
 Derived data
 introduced for performance 10-30T
 produced by view 11-20T
 DESC keyword 1-338S
 syntax
 in CREATE INDEX 1-63S
 in SELECT 1-337S
 use
 in CREATE INDEX 1-66S
 in SELECT 1-338S
 Descending order in SELECT 2-14T
 DESCRIBE statement
 and COLLCHAR environment
 variable 4-46R
 and the USING SQL DESCRIPTOR
 clause 1-164S
 describing statement type 1-163S, 5-33T
 INTO sqllda pointer clause 1-165S
 relation to GET DESCRIPTOR 1-213S
 syntax 1-162S
 using with concatenation operator 1-432S
 values returned by SELECT 1-163S
 Descriptor 1-164S
 Determining ANSI-compliance 1-10R
 Device
 optical 12-6T
 storage 12-6T
 Directory, extension, .dbs 1-58S, 1-141S
 Dirty Read isolation level 1-367S, 7-10T
 DIRTY READ keywords, syntax in SET ISOLATION 1-366S
 DISCONNECT statement 1-169S
 ALL keyword 1-169S
 CURRENT keyword 1-168S
 with
 concatenation operator 1-432S
 Disk access
 chunk 10-4T

-
- cost to read a row 13-16T
 - dbspace 10-4T
 - latency of 13-17T
 - nonsequential 13-18T
 - nonsequential avoided by sorting 13-37T
 - performance 13-17T to 13-19T, 13-34T
 - reducing contention 10-7T
 - seek time 13-17T
 - sequential 13-17T, 13-34T
 - sequential forced by query 13-32T, 13-33T, 13-35T
 - using rowid 13-18T
 - Disk buffer. *See* Page buffer.
 - Disk contention
 - effect of 13-17T
 - multiple arms to reduce 10-7T
 - Disk extent 10-8T
 - Disk mirroring 10-5T
 - Disk page
 - buffer for 13-16T, 13-18T
 - size of 10-4T, 13-14T
 - Display label
 - in ORDER BY clause 2-52T
 - syntax in SELECT 1-312S
 - with SELECT 2-49T
 - Display schema for a database 5-34R
 - DISTINCT keyword
 - relation to GROUP BY 3-5T
 - restrictions in modifiable view 11-24T
 - syntax
 - in CREATE INDEX 1-63S
 - in expression 1-462S
 - in SELECT 1-312S
 - use
 - in CREATE INDEX 1-64S
 - in SELECT 1-313S, 2-19T
 - no effect in subquery 1-414S
 - with COUNT function 2-54T
 - Distributed deadlock 7-15T
 - Distributed processing 12-8T
 - Distributions
 - dropping with DROP DISTRIBUTIONS clause 1-396S
 - privileges required to create 1-396S
 - using the HIGH keyword 1-396S
 - using the MEDIUM keyword 1-397S
 - Division (/) symbol, arithmetic operator 1-431S
 - DML statements, summary 1-6S
 - DOCUMENT keyword, use in stored procedures 14-6T
 - Documentation notes Intro-8R, Intro-11S, 6T
 - Dominant table 3-21T
 - Dormant connection 1-169S
 - DOS operating system 12-4T
 - DOUBLE PRECISION data type. *See* FLOAT data type.
 - DROP AUDIT statement 1-171S
 - DROP CONSTRAINT keywords
 - syntax in ALTER TABLE 1-15S
 - use in ALTER TABLE 1-32S
 - DROP DATABASE statement 1-172S
 - DROP INDEX statement
 - locks table 7-7T
 - releasing an index 13-34T
 - syntax 1-174S
 - DROP keyword
 - syntax in ALTER TABLE 1-15S
 - use in ALTER TABLE 1-26S
 - DROP SYNONYM statement 1-177S
 - DROP TABLE statement 1-179S
 - DROP TRIGGER statement
 - syntax 1-181S
 - use of 1-181S
 - DROP VIEW statement 1-183S
 - Duplicate index keys 10-22T
 - Duplicate values
 - finding 3-16T
 - in a query 1-313S
 - Dynamic management statements 1-6S
 - Dynamic SQL
 - cursor use with 5-32T
 - description of 5-5T, 5-28T
 - freeing prepared statements 5-33T
- ## E
- Editor, specifying with DBEDIT 4-18R
 - Effective checking 1-349S
 - Ellipses (...), wildcard in Condition segment 1-411S
 - Embedded SQL
 - defined 5-3T
 - languages available 5-4T
 - See also* ESQL.

End of data
 signal in SQLCODE 5-11T, 5-17T
 signal only for SELECT 6-14T
 when opening cursor 5-20T

Entity
 attributes associated with 8-15T
 business rules 8-5T
 criteria for choosing 8-7T
 defined 8-4T
 important qualities of 8-5T
 in telephone-directory example 8-7T
 integrity 4-18T
 naming 8-4T
 represented by a table 8-22T

Entity occurrence, defined 8-16T

Entity-relationship diagram
 connectivity 8-17T
 discussed 8-17T
 meaning of symbols 8-17T
 reading 8-18T

ENVIGNORE environment variable
 4-30R
 relation to chkenv utility 5-4R

Environment configuration file
 debugging with chkenv 5-4R
 example 4-4R
 where stored 4-5R

Environment variable
 and case sensitivity 4-6R
 ARC_DEFAULT 4-14R
 ARC_KEYPAD 4-14R
 COLLCHAR 4-45R
 DBANSIWARN 4-15R
 DBAPICODE 4-47R
 DBAPICODE, and crtcmap utility
 5-5R
 DBDATE 4-16R
 DBDELIMITER 4-18R
 DBEDIT 4-18R
 DBLANG 4-19R
 DBLANG, and crtcmap utility 5-5R
 DBMONEY 4-20R
 DBNLS 4-48R
 DBPATH 4-21R, 12-14T
 DBPRINT 4-24R
 DBREMOTECMD 4-25R
 DBSPACETEMP 4-26R
 DBTEMP 4-27R
 DBTIME 4-27R
 DBUPSPACE 4-30R
 default assumptions 4-6R
 defining in environment
 configuration file 4-4R
 definition of 4-3R
 ENVIGNORE 4-30R
 ENVIGNORE, and chkenv utility
 5-4R
 how to set in Bourne shell 4-5R
 how to set in C shell 4-5R
 how to set in Korn shell 4-5R
 INFORMIX environment variables,
 listing 4-8R
 INFORMIXC 4-31R
 INFORMIXCOB 4-32R
 INFORMIXCOBDIR 4-32R
 INFORMIXCOBSTORE 4-33R
 INFORMIXCONRETRY 4-34R
 INFORMIXCONTIME 4-34R
 INFORMIXDIR 4-36R, 12-14T
 INFORMIXSERVER 4-36R, 12-14T
 INFORMIXSHMBASE 4-37R
 INFORMIXSTACKSIZE 4-38R
 INFORMIXTERM 4-38R
 LANG 4-49R
 LANG, and crtcmap utility 5-5R
 LC_COLLATE 4-50R
 LC_CTYPE 4-51R
 LC_MONETARY 4-52R
 LC_NUMERIC 4-53R
 LC_TIME 4-53R
 listed 4-8R
 listed, by topic 4-9R
 listed, for NLS 4-9R
 listed, for UNIX 4-9R
 NLS environment variables, listing
 4-9R
 ONCONFIG 4-39R
 overriding a setting 4-4R, 4-30R
 PATH 4-54R, 12-14T
 PSORT_DBTEMP 4-40R
 PSORT_NPROCS 4-41R
 rules of precedence 4-7R
 setting at the command line 4-4R
 setting in a shell file 4-4R
 SQLEXEC 4-41R
 SQLRM 4-41R, 4-42R
 SQLRMDIR 4-43R
 TERM 4-55R
 TERMCAP 4-56R, 12-14T
 TERMINFO 4-56R
 UNIX environment variables, listing
 4-9R
 where to set 4-4R

- Equals (=) relational operator 2-30T, 2-71T
- Equi-join 2-71T
- ERROR 1-400S
- Error checking
 - continuing after error in stored procedure 2-34S
 - error status with ON EXCEPTION 2-32S
 - exception handling 14-27T
 - in stored procedures 14-27T
 - simulating errors 14-31T
 - with SYSTEM 2-40S
- ERROR keyword, in the WHENEVER statement 1-398S
- Error messages
 - for NLS 1-21R
 - for trigger failure 15-14T
 - generating in a trigger 15-14T
 - retrieving trigger text in a program 15-16T, 15-17T
- Errors
 - after DELETE 6-4T
 - at compile time 14-6T
 - codes for 5-11T
 - dealing with 5-16T
 - detected on opening cursor 5-20T
 - during updates 4-22T
 - in stored procedure syntax 14-7T
 - inserting with a cursor 6-11T
 - ISAM error code 5-12T
 - using to identify NLS database server 1-19R
- ESCAPE keyword
 - syntax in Condition segment 1-405S
 - use
 - in Condition segment 1-409S
 - with LIKE keyword 1-328S, 1-411S
 - with MATCHES keyword 1-329S, 1-412S
 - with WHERE keyword 1-328S, 2-43T
- ESQL
 - cursor use 5-19T to 5-28T
 - DELETE statement in 6-3T
 - delimiting host variables 5-6T
 - dynamic embedding 5-5T, 5-28T
 - error handling 5-16T
 - fetching rows from cursor 5-21T
 - host variable 5-6T, 5-8T
 - indicator variable 5-15T
 - INSERT in 6-8T
 - NLS errors in SQLERRM field 1-19R
 - overview 5-3T to 5-37T, 6-3T to 6-17T
 - preprocessor 5-4T
 - scroll cursor 5-23T
 - selecting single rows 5-13T
 - SQL Communications Area 5-8T
 - SQLCODE 5-11T
 - SQLERRD fields 5-12T
 - static embedding 5-5T
 - UPDATE in 6-14T
- Estimating
 - blobpages 10-17T
 - maximum number of extents 10-10T
 - size of index 10-16T
 - table size with fixed-length rows 10-13T
 - table size with variable-length rows 10-14T
- Example database. *See* Demonstration database.
- EXCLUSIVE keyword
 - syntax
 - in DATABASE 1-140S
 - in LOCK TABLE 1-260S
 - use
 - in DATABASE 1-142S
 - in LOCK TABLE 1-262S
- Exclusive lock 7-6T
- EXECUTE IMMEDIATE statement
 - description of 5-34T
 - restricted statement types 1-191S
 - syntax and usage 1-190S
 - using with concatenation operator 1-432S
- EXECUTE ON keywords
 - syntax
 - in GRANT 1-231S
 - in REVOKE 1-300S
 - use
 - in GRANT 1-237S
 - in REVOKE 1-300S
- EXECUTE PROCEDURE statement
 - 1-192S
 - assigning values with 14-21T
 - associating cursor with DECLARE 1-148S
 - in FOREACH 2-20S
 - in triggered action 1-122S
 - using 14-9T

- with
 - INTO keyword 1-198S
- EXECUTE statement
 - and sqlca record 1-186S
 - description of 5-31T
 - parameterizing a statement 1-186S
 - syntax 1-184S
 - USING DESCRIPTOR clause 1-188S
 - with concatenation operator 1-432S
 - with USING keyword 1-186S
- Existence dependency 8-9T
- EXISTS keyword
 - beginning a subquery 1-330S
 - in a WHERE clause 3-33T
 - use in condition subquery 1-414S, 11-27T
- EXIT statement
 - exiting a loop 14-23T
 - syntax 2-14S
- EXP function
 - syntax in expression 1-451S
 - use in expression 1-451S
- Explicit temporary table 1-100S
- Exponential function
 - EXP function 1-451S
- Exponential function. *See* EXP function.
- Expression
 - date-oriented 2-56T
 - description of 2-46T
 - display label for 2-49T
 - in UPDATE 1-387S
 - ordering by 1-339S
- Expression segment
 - aggregate expressions 1-462S
 - column expressions 1-433S
 - combined expressions 1-467S
 - constant expressions 1-436S
 - expression types 1-431S
 - function expressions 1-443S
 - in SPL expressions 1-468S
 - syntax 1-431S
- EXTEND function
 - syntax in expression 1-454S
 - use in expression 1-456S
 - with DATE, DATETIME and INTERVAL 3-26R, 3-28R, 2-56T, 2-61T
- extension
 - .dat Intro-11R

- .dbs Intro-11R
- .idx Intro-11R
- .lok 1-370S
- Extension checking, specifying with DBANSIWARN 4-15R
- Extension, to SQL
 - symbol for Intro-7S
 - with ANSI-compliant database 1-14R
- Extent
 - changing size of for system table 2-9R
 - description of 10-8T
 - next extent doubles 10-10T
 - sizes of 10-8T
 - upper limit on 10-10T
- EXTENT SIZE keywords 1-107S, 10-8T

F

- FETCH statement
 - ABSOLUTE keyword 5-23T
 - as affected by CLOSE 1-39S
 - checking results with SQLCA 1-202S
 - description of 5-21T
 - fetching a row for update 1-201S
 - locking for update 1-201S
 - relation to GET DESCRIPTOR 1-210S
 - sequential 5-23T
 - specifying a value's memory location 1-198S
 - syntax 1-194S
 - with
 - concatenation operator 1-432S
 - INTO keyword 1-319S
 - program arrays 1-199S
 - scroll cursor 1-196S
 - sequential cursor 1-195S, 5-25T
 - X/Open mode 1-195S
- Field qualifier
 - for DATETIME 3-8R, 1-428S
 - for INTERVAL 3-13R, 1-485S, 1-491S
- File
 - compared to database 1-3T
 - environment configuration 4-4R
 - environment configuration, checking with chkenv 5-4R
 - extension
 - .lok 1-370S
 - mapping, and COLLCHAR 4-45R
 - mapping, and crtcmap utility 5-5R
 - mapping, and DBAPICODE 4-47R
 - mapping, format for 5-6R

- permissions in UNIX 11-4T
- sending output with the OUTPUT statement 1-271S
- shell 4-4R
- temporary for OnLine 4-26R
- temporary for SE 4-27R
- FILE statement
 - character-position form 5-28R
 - delimiter form 5-24R
 - syntax for character-position form 5-29R
 - syntax for delimiter form 5-25R
 - with dbload 5-23R
- Filter expression
 - effect on performance 13-23T, 13-32T, 13-33T
 - evaluated from index 13-10T, 13-31T
 - optimizer uses 13-10T, 13-23T
 - selectivity estimates 13-30T
- finding location of row 1-435S
- FIRST keyword
 - syntax in FETCH 1-194S
 - use in FETCH 1-196S
- First normal form 8-29T
- Fixed point 9-10T
- FLOAT data type
 - changing data types 3-23R
 - description of 3-12R, 9-9T
 - syntax 1-425S
 - using as default value 1-19S, 1-89S
- Floating point 9-9T
- FLUSH statement
 - count of rows inserted 6-11T
 - syntax 1-204S
 - with concatenation operator 1-432S
 - writing rows to buffer 6-10T
- FOR EACH ROW action
 - SELECT statement in 1-117S
 - triggered action list 1-116S
- FOR EACH ROW action. *See also* FOREACH keyword.
- FOR keyword
 - in CONTINUE 2-6S
 - in CREATE AUDIT 1-55S
 - in CREATE SYNONYM 1-80S
 - in EXIT 2-14S
- for locating temporary tables 1-100S
- FOR statement
 - looping in a stored procedure 14-23T
 - specifying multiple ranges 2-18S
 - syntax 2-16S
 - using expression lists 2-18S
 - using increments 2-17S
- FOR TABLE keywords, in UPDATE STATISTICS 1-393S
- FOR UPDATE keywords
 - conflicts with ORDER BY 6-8T
 - not needed in ANSI-compliant database 6-16T
 - relation to UPDATE 1-390S
 - specific columns 6-16T
 - syntax in DECLARE 1-145S
 - use
 - in DECLARE 1-148S, 1-152S, 1-155S
 - in SELECT 1-340S
 - with column list 1-153S
- FOREACH keyword
 - in CONTINUE statement 2-6S
 - in EXIT 2-14S
- FOREACH keyword. *See also* FOREACH ROW action.
- FOREACH statement
 - looping in a stored procedure 14-23T
 - syntax 2-20S
- Foreign characters, using NLS 1-14R
- Foreign key 1-22S, 1-93S, 1-94S, 4-19T
 - See also* Referential constraint.
- FOREIGN KEY keywords
 - in ALTER TABLE 1-29S
 - in CREATE TABLE 1-92S
- Format
 - for crtmap mapping file 5-6R
 - specifying for DATE value with DBDATE 4-16R
 - specifying for DATETIME value with DBTIME 4-27R
 - specifying for MONEY value with DBMONEY 4-20R
- FRACTION keyword
 - syntax
 - in DATETIME data type 1-428S
 - in INTERVAL data type 1-485S
 - use
 - as DATETIME field qualifier 3-8R, 1-488S
 - as INTERVAL field qualifier 3-14R, 1-491S
- FREE statement

- effect on cursors 1-269S
- freeing prepared statements 5-33T
- syntax 1-207S
- with concatenation operator 1-432S

FROM keyword

- alias names 2-79T
- syntax
 - in PUT 1-284S
 - in REVOKE 1-300S
 - in SELECT 1-310S
- use
 - in PUT 1-287S
 - in SELECT 1-323S

Function

- aggregate 2-53T
- algebraic 1-444S
- date-oriented 2-56T
- in SELECT statements 2-53T
- within a stored procedure 14-24T

Function expression

- DBINFO function 1-448S
- description of 1-443S
- in SELECT 1-314S

Functional dependency 8-31T

G

GET DESCRIPTOR statement

- syntax 1-210S
- the COUNT keyword 1-212S
- use with FETCH statement 1-200S
- with concatenation operator 1-432S
- X/Open mode 1-213S

GET DIAGNOSTICS statement

- CLASS_ORIGIN keyword 1-225S
- CONNECTION_NAME keyword 1-227S
- exception clause 1-223S
- MESSAGE_LENGTH keyword 1-225S
- MESSAGE_TEXT keyword 1-225S
- MORE keyword 1-222S
- NUMBER keyword 1-223S
- purpose 1-217S
- RETURNED_SQLSTATE keyword 1-225S
- ROW_COUNT keyword 1-223S
- SERVER_NAME keyword 1-225S
- statement clause 1-222S
 - keywords 1-222S
- SUBCLASS_ORIGIN keyword 1-225S

- syntax 1-217S

Global transaction 12-21T

GOTO keyword, in the WHENEVER statement 1-398S

GRANT statement

- automated 11-13T
- changing grantor 1-238S
- creating a privilege chain 1-238S
- database-level privileges 1-232S, 11-5T
- default table privileges 1-237S
- in 4GL 11-13T
- in embedded SQL 5-34T to 5-37T
- passing grant ability 1-237S
- privileges on a view 1-239S
- syntax 1-231S
- table-level privileges 1-235S, 11-7T
- use of dbschema 5-34R
- with CREATE SCHEMA 1-77S

GROUP BY clause 1-125S

GROUP BY keywords

- column number with 3-7T
- composite index used for 13-31T
- description of 3-4T
- indexes for 13-10T, 13-34T
- restrictions in modifiable view 11-24T
- sorting rows 13-12T
- syntax in SELECT 1-310S
- use in SELECT 1-334S

H

HAVING keyword

- description of 3-9T
- syntax in SELECT 1-310S
- use in SELECT 1-336S

Header, of a procedure 14-25T

HEX function 1-435S

- syntax in expression 1-452S
- use in expression 1-452S

HIGH keyword 1-374S

Hold cursor

- definition of 1-149S, 7-18T
- insert cursor with hold 1-157S
- use of 1-150S

Host machine 12-8T

Host variable

- delimiter for 5-6T
- description of 5-6T
- dynamic allocation of 5-33T

sharing with constraints 1-86S
 sorting with NLS 1-15R
 time cost of 10-20T
 updating affects 13-31T
 utility to test or repair 13-31T
 when not used by optimizer 13-32T, 13-33T, 13-35T
 with temporary tables 1-343S
INDEX keyword
 syntax
 in GRANT 1-235S
 in REVOKE 1-301S
 use
 in GRANT 1-236S
 in REVOKE 1-302S
Index Name segment
 syntax 1-419S, 1-495S
 use 1-504S
Index privilege 1-236S, 11-8T
INDEXES FOR keywords, in **INFO** statement 1-242S
Indexes, with dbload utility 5-21R
INDICATOR field
 setting with SET DESCRIPTOR 1-358S
INDICATOR keyword, in **SELECT** 1-319S, 1-320S, 1-321S, 1-432S
Indicator variable
 definition of 5-15T
 in EXECUTE 1-186S
 in expression 1-467S
 in **SELECT** 1-319S, 1-320S, 1-321S, 1-432S
 using non-English characters in 1-22R
INFO statement
 displaying privileges and status 1-242S
 displaying tables, columns, and indexes 1-241S
 syntax 1-241S
informix
 environment configuration file 4-4R
 privileges associated with user 1-234S
Informix extension checking, specifying with DBANSIWARN 4-15R
INFORMIX-4GL 12-5T
 detecting null value 5-16T
 example of dynamic SQL 11-13T
 indicator variable not used 5-16T
 program variable 5-5T
 STATUS variable 5-11T
 terminates on errors 5-36T, 6-14T
 timing operations in 13-6T
 using SQLCODE with 5-11T
 WHENEVER ERROR statement 5-36T
INFORMIXC environment variable 4-31R
INFORMIXCOB environment variable 4-32R
INFORMIXCOBDIR environment variable 4-32R
INFORMIXCOBSTORE environment variable 4-33R
INFORMIXCOBTYPE environment variable 4-33R
INFORMIXCONRETRY environment variable 4-34R
INFORMIXCONTIME environment variable 4-34R
INFORMIXDIR environment variable 4-36R, 12-14T
INFORMIX-OnLine
 disk access by 13-17T
INFORMIX-OnLine Dynamic Server
 allows views on external tables 11-23T
 and triggering statement 1-112S
 archiving 4-26T
 characteristics of 1-10T
 creating demonstration database Intro-10R, Intro-13S
 disk page size 13-16T
 disk storage methods 10-3T to 10-12T
 is NLS-ready 1-15R
 optimizer input with 13-9T
 signalled in SQLAWARN 5-12T
 when tables are locked 7-7T
INFORMIX-OnLine/Optical 12-6T
 list of statements 1-7S
INFORMIX-SE
 characteristics of 1-10T
 creating database 9-23T
 creating demonstration database Intro-10R, Intro-13S
 is NLS-ready 1-15R
INFORMIXSERVER environment variable 4-36R, 1-46S, 12-14T
INFORMIXSHMBASE environment variable 4-37R
INFORMIX-SQL

- creating database with 5-34T, 9-26T
 - UNLOAD statement 9-27T
- INFORMIXSTACKSIZE environment variable 4-38R
- INFORMIXTERM environment variable 4-38R
- informix.rc file 4-4R
- Insert buffer
 - counting inserted rows 1-205S, 1-290S
 - filling with constant values 1-286S
 - inserting rows with a cursor 1-248S
 - storing rows with PUT 1-285S
 - triggering flushing 1-289S
- Insert cursor 1-148S
 - closing 1-39S
 - definition of 1-148S, 6-8T
 - in INSERT 1-248S
 - in PUT 1-286S
 - opening 1-266S
 - reopening 1-267S
 - result of CLOSE in SQLCA 1-39S
 - use of 1-149S, 6-11T
 - with hold 1-157S
 - writing buffered rows with FLUSH 1-204S
- INSERT INTO keywords
 - in INSERT 1-245S
 - in LOAD 1-258S
- INSERT keyword
 - syntax
 - in GRANT 1-235S
 - in REVOKE 1-301S
 - use
 - in GRANT 1-236S
 - in REVOKE 1-302S
- Insert privilege 11-8T, 11-28T
- INSERT REFERENCING clause
 - and FOR EACH ROW section 1-121S
 - correlation name 1-118S
 - syntax 1-118S
- INSERT STATEMENT
 - using functions in the VALUES clause 1-252S
- INSERT statement
 - and end of data 6-14T
 - character-position form 5-28R
 - constant data with 6-11T
 - count of rows inserted 6-11T
 - delimiter form 5-24R
 - duplicate values in 4-7T
 - effect of transactions 1-249S
 - embedded 6-8T to 6-14T
 - filling insert buffer with PUT 1-285S
 - in dynamic SQL 1-253S
 - in trigger event 1-111S
 - in triggered action 1-122S
 - inserting
 - multiple rows 4-9T
 - nulls with the VALUES clause 1-252S
 - rows 4-6T
 - rows through a view 1-246S
 - rows with a cursor 1-248S
 - single rows 4-7T
 - values into SERIAL columns 3-19R
 - null values in 4-7T
 - number of rows 5-12T
 - privilege for 11-6T, 11-8T
 - SELECT statement in 4-9T
 - SERIAL columns 1-251S
 - specifying values to insert 1-250S
 - syntax 1-245S
 - for character position form 5-29R
 - for delimiter form 5-25R
 - time to update indexes 10-20T
 - use with insert cursor 1-157S
 - VALUES clause 4-7T
 - with
 - a view 11-25T
 - DECLARE 1-145S
 - SELECT 1-253S
 - with dbload 5-23R
- Inserting rows of constant data 6-11T
- Installation directory, specifying with INFORMIXDIR 4-36R
- INT data type. *See* INTEGER data type.
- INTEGER data type
 - changing data types 3-23R
 - description of 3-13R, 9-7T
 - syntax 1-425S
 - using as default value 1-19S, 1-89S
- Integrity. *See* Data integrity.
- Integrity. *See* Referential integrity.
- Integrity. *See* Semantic integrity.
- Intensity attributes, setting INFORMIXTERM for 4-38R
- Interrupt key
 - with dbexport 5-9R
 - with dbimport 5-14R
 - with dbload 5-21R

-
- Interrupted modifications 4-22T
 - INTERVAL data type
 - adding or subtracting from 3-30R
 - adding or subtracting from DATETIME values 3-27R
 - as quoted string 1-498S
 - description of 3-13R, 9-14T
 - display format 9-16T
 - field delimiters 3-15R
 - field qualifier, syntax 1-485S
 - in expression 1-441S
 - in INSERT 1-498S
 - in relational expressions 2-29T
 - multiplying or dividing values 3-31R
 - precision and size 9-14T
 - range of expressions 3-26R
 - range of operations with DATE and DATETIME 3-25R
 - syntax 1-425S, 1-491S
 - with EXTEND function 3-26R, 3-28R
 - See also* Literal INTERVAL.
 - INTERVAL Field Qualifier segment 1-485S
 - INTO keyword
 - choice of location 5-22T
 - in FETCH statement 5-22T
 - in SELECT 1-318S
 - mismatch signalled in SQLAWARN 5-12T
 - restrictions in INSERT 4-10T
 - restrictions in prepared statement 5-29T
 - retrieving multiple rows 5-20T
 - retrieving single rows 5-13T
 - syntax
 - in FETCH 1-194S
 - in SELECT 1-310S
 - use
 - in FETCH 1-199S
 - in SELECT 1-318S
 - INTO TEMP keywords
 - description of 2-83T
 - restrictions in view 11-22T
 - syntax in SELECT 1-310S
 - use
 - in SELECT 1-341S
 - with UNION operator 1-344S
 - IS keyword
 - in Condition segment 1-409S
 - with WHERE keyword 1-327S
 - IS NOT keywords, syntax in Condition segment 1-405S
 - IS NULL keywords 1-327S
 - ISAM error code 2-31S, 2-36S, 5-12T
 - Isolation level
 - Committed Read 1-367S, 7-10T
 - Cursor Stability 1-367S, 7-11T
 - default in ANSI-compliant database 1-13R
 - definitions 1-367S
 - description of 7-9T
 - Dirty Read 1-367S, 7-10T
 - in external tables 1-368S
 - Repeatable Read 1-367S, 7-12T
 - setting 7-9T
 - use with FETCH statement 1-201S
 - items table in stores6 database, columns in A-3R
 - ITYPE field
 - SET DESCRIPTOR statement 1-357S
 - setting with SET DESCRIPTOR 1-359S
 - with X/Open programs 1-213S
- ## J
- Join
 - associative 2-76T
 - creating 2-71T
 - definition of 2-8T
 - dominant table 3-21T
 - effect of large join on optimization 13-36T
 - equi-join 2-71T
 - in Condition segment 1-331S
 - multiple-table join 1-332S, 2-77T
 - natural 2-75T
 - nested outer 3-28T
 - nested simple 3-25T
 - outer 3-21T
 - outer join 1-333S
 - restrictions in modifiable view 11-24T
 - self-join 1-333S, 3-11T
 - sort merge 13-27T
 - subservient table 3-21T
 - two-table join 1-332S
 - Join column. *See* Foreign key.
 - Join condition. *See* Condition segment.
 - Journal updates 10-33T

K

Key lock 7-8T
Keywords, using in triggered action
1-123S
Key, candidate. *See* Candidate key.
Key, composite 8-23T
Key, foreign. *See* Foreign key.
Key, primary 8-22T
Key, primary. *See* Primary key
constraint.
Korn shell
 how to set environment variables
 4-5R
 .profile file 4-4R

L

Label 2-49T, 3-48T
LANG environment variable 1-16R,
4-49R
Language supplement for added NLS
functionality 1-22R
LAST keyword
 syntax in FETCH 1-194S
 use in FETCH 1-196S
Latency 13-17T
LC_COLLATE environment variable
1-16R, 4-50R
LC_CTYPE environment variable 1-16R,
4-51R
LC_MONETARY environment variable
1-17R, 4-52R
LC_NUMERIC environment variable
1-17R, 4-53R
LC_TIME environment variable 1-17R,
4-53R
LENGTH field
 setting with SET DESCRIPTOR 1-357S
 with DATETIME and INTERVAL
 types 1-358S
 with DECIMAL and MONEY types
 1-358S
LENGTH function
 in expression 1-314S
 on TEXT 2-64T
 on VARCHAR 2-64T
 syntax in expression 1-443S, 1-453S
 use in expression 1-453S, 2-63T

LET statement
 assigning values 14-21T
 executing a procedure 14-9T
 syntax 2-28S
LIKE keyword
 syntax in Condition segment 1-405S
 use in SELECT 1-328S
 wildcard characters 1-328S
LIKE keyword, used to test for equality
 in WHERE clause 2-29T
LIKE relational operator 2-37T
LIKE test 13-32T
Literal
 DATETIME
 in Condition segment 1-405S
 in expression 1-436S, 1-441S
 segment 1-487S
 syntax 1-488S
 syntax in INSERT 1-250S
 use in ALTER TABLE 1-18S
 use in CREATE TABLE 1-88S
 with IN keyword 1-327S
 DATE, using as a default value 1-19S,
 1-89S
 INTERVAL
 in Condition segment 1-405S
 in expression 1-436S, 1-441S
 segment 1-490S
 syntax 1-491S
 syntax in INSERT 1-250S
 using as default value 1-19S, 1-89S
 Number
 in Condition segment 1-405S
 in expression 1-436S, 1-439S
 segment 1-493S
 syntax 1-493S
 syntax in INSERT 1-250S
 with IN keyword 1-409S
LOAD statement
 DELIMITER clause 1-258S
 input formats for data 1-256S
 INSERT INTO clause 1-258S
 loading VARCHAR, TEXT, or BYTE
 data types 1-257S
 specifying field delimiter with
 DBDELIMITER 4-18R
 specifying the table to load into 1-258S
 syntax 1-255S
 the LOAD FROM file 1-256S
Loading data from a command file into a
table 5-20R

Local loopback 12-9T
 Local server 12-5T
 LOCK MODE keywords
 syntax
 in ALTER TABLE 1-15S
 in CREATE TABLE 1-108S
 use
 in ALTER TABLE 1-33S
 in CREATE TABLE 1-108S
 LOCK TABLE statement
 in databases with transactions 1-261S
 in databases without transactions 1-262S
 locking a table explicitly 7-7T
 syntax 1-260S
 Locking
 and concurrency 4-27T
 and integrity 7-3T
 deadlock 7-14T
 description of 7-5T
 during
 delete 1-159S
 inserts 1-249S
 updates 1-152S, 1-385S
 granularity 7-6T
 in OnLine 1-5R
 in SE 1-6R
 lock duration 7-8T
 lock mode 7-13T
 not-wait 7-14T
 wait 7-13T
 locks released at end of transaction 7-18T
 mode 1-5R
 overriding row-level 1-261S
 releasing with COMMIT WORK 1-43S
 releasing with ROLLBACK WORK 1-306S
 scope 1-5R, 7-6T
 scope of lock 7-6T
 setting lock mode 7-13T
 shared locks 1-6R
 types of locks
 database lock 7-6T
 exclusive lock 7-6T
 key lock 7-8T
 page lock 1-108S, 7-8T
 promotable lock 7-6T, 7-9T
 row lock 1-108S, 7-8T
 shared lock 7-6T
 table lock 7-7T
 update cursors effect on 1-152S
 update locks 1-385S
 waiting period 1-370S
 with
 DELETE 6-4T
 FETCH 1-201S
 scroll cursor 1-369S
 SET ISOLATION 1-366S
 SET LOCK MODE 1-370S
 UNLOCK TABLE 1-381S
 update cursor 7-9T
 within transaction 1-35S
 See also Table locking.
 LOG IN keywords, syntax in CREATE DATABASE 1-57S
 LOG10 function
 syntax in expression 1-452S
 use in expression 1-452S
 Logarithmic function
 syntax
 LOG10 function 1-451S
 LOGN function 1-451S
 use
 LOG10 function 1-452S
 LOGN function 1-452S
 Logging
 buffered 9-22T
 buffered vs. unbuffered 1-372S
 cascading deletes 1-24S, 1-98S
 changing mode with SET LOG 1-372S
 choosing for OnLine database server 9-22T
 choosing for SE database server 9-23T
 finding log file location 1-60S
 renaming log 1-377S
 setting with CREATE TABLE 1-104S
 starting with START DATABASE 1-60S, 1-376S
 stopping 1-377S
 stopping with START DATABASE 1-376S
 unbuffered 9-22T
 with INFORMIX-OnLine 1-59S
 with INFORMIX-SE 1-60S
 with triggers 1-134S
 Logical log
 definition of 4-25T
 Logical operator
 AND 2-36T
 in Condition segment 1-417S
 NOT 2-36T

-
- OR 2-36T
 - LOGN function
 - syntax in expression 1-452S
 - use in expression 1-452S
 - .lok extension 1-370S
 - Loop
 - controlled 2-16S
 - creating and exiting in SPL 14-23T
 - exiting using RAISE exception 14-32T
 - indefinite with WHILE 2-46S
 - Loopback, local 12-9T
 - LOW keyword 1-374S
- M**
- Machine notes Intro-9R, Intro-11S, 6T
 - Mandatory, entity in relationship 8-9T
 - Many-to-many relationship 8-9T, 8-11T, 8-26T
 - Mapping files for non-standard code sets 5-5R
 - MATCHES keyword
 - syntax in Condition segment 1-405S
 - use
 - in Condition segment 1-409S
 - in SELECT 1-328S
 - with NLS 2-42T
 - used to test for equality in WHERE clause 2-29T
 - wildcard characters 1-329S
 - MATCHES relational operator
 - in WHERE clause 2-37T
 - MAX function
 - as aggregate function 2-53T
 - syntax in expression 1-462S
 - use in expression 1-464S
 - MDY function
 - as time function 2-56T
 - syntax in expression 1-454S
 - Memory
 - allocating for a system sqllda structure 1-9S
 - shared 12-7T
 - Message files
 - error messages Intro-9R, Intro-11S, 7T
 - setting LANG for NLS 4-49R
 - specifying subdirectory for NLS 1-22R
 - specifying subdirectory with DBLANG 4-19R
 - MIN function
 - as aggregate function 2-53T
 - syntax in expression 1-462S
 - use in expression 1-464S
 - Minus (-) sign, arithmetic operator 1-431S
 - MINUTE keyword
 - syntax
 - in DATETIME data type 1-428S
 - in INTERVAL data type 1-485S
 - use
 - as DATETIME field qualifier 3-8R, 1-488S
 - as INTERVAL field qualifier 3-14R, 1-491S
 - Mirror. *See* Disk mirroring.
 - MOD function
 - syntax in expression 1-444S
 - use in expression 1-446S
 - MODE ANSI keywords
 - ANSI-compliant database 1-15T
 - ANSI-compliant logging 9-23T
 - specifying ANSI-compliance 1-11R
 - specifying transactions 4-24T
 - syntax
 - in CREATE DATABASE 1-57S
 - in START DATABASE 1-376S
 - use
 - in CREATE DATABASE 1-60S
 - in START DATABASE 1-377S
 - Model. *See* Data model.
 - MODIFY keyword
 - syntax in ALTER TABLE 1-15S
 - use in ALTER TABLE 1-27S
 - MODIFY NEXT SIZE keywords
 - syntax in ALTER TABLE 1-15S
 - use in ALTER TABLE 1-33S
 - MONEY data type 9-12T
 - changing data types 3-23R
 - description of 3-16R, 9-10T
 - display format 9-12T
 - in INSERT 4-8T
 - syntax 1-425S
 - using as default value 1-19S, 1-89S
 - See also* DECIMAL data type.
 - MONEY value
 - setting DBMONEY environment variable 4-20R
 - setting LC_MONETARY environment variable 4-52R

-
- specifying European format with DBMONEY 4-20R
 - specifying for NLS 4-53R
 - Money, representing with NLS 1-15R
 - MONTH function
 - as time function 2-56T
 - syntax in expression 1-454S
 - use in expression 1-456S
 - MONTH keyword
 - syntax
 - in DATETIME data type 1-428S
 - in INTERVAL data type 1-485S
 - use
 - as DATETIME field qualifier 3-8R, 1-488S
 - as INTERVAL field qualifier 3-13R, 1-491S
 - Multiple triggers
 - column numbers in 1-115S
 - example 1-114S
 - order of execution 1-115S
 - preventing overriding 1-132S
 - Multiple-table join 2-77T
 - Multi-row query
 - destination of returned values 1-198S
 - managing with FETCH 1-195S
 - N**
 - Naming convention
 - column 1-87S, 1-102S, 1-294S
 - database 1-422S
 - index 1-419S, 1-484S, 1-505S
 - table 1-84S, 1-470S, 1-507S
 - See also* Database Name segment.
 - See also* Index Name segment.
 - See also* Table Name segment.
 - Naming conventions
 - tables 12-16T
 - Native Language Support. *See* NLS.
 - Natural join 2-75T
 - NCHAR data type
 - description of 3-17R, 9-15T
 - syntax 1-425S
 - versus CHAR data type 1-19R
 - Nested ordering, in SELECT 1-339S, 2-15T
 - Network
 - computer 12-4T
 - connection information 12-15T
 - data sent over 13-21T
 - performance of 13-20T
 - simple model of 13-21T
 - site 12-4T
 - Network environment variable
 - DBPATH 4-21R
 - SQLRM 4-42R
 - SQLRMDIR 4-43R
 - NEW keyword
 - in DELETE REFERENCING clause 1-119S
 - in INSERT REFERENCING clause 1-118S
 - in UPDATE REFERENCING clause 1-120S
 - NEXT keyword
 - syntax in FETCH 1-194S
 - use in FETCH 1-196S
 - NEXT SIZE keywords
 - specifying size of extents 10-8T
 - use in CREATE TABLE 1-107S
 - use in GRANT 1-233S
 - NLS 1-14R
 - activating 2-25T
 - activating in Informix products 1-16R
 - and collation order 2-25T
 - and dbexport utility 5-12R
 - and dbimport utility 5-13R
 - and MATCHES keyword 2-42T
 - and ORDER BY keywords 2-25T, 2-42T
 - checking products for functionality 1-18R
 - COLLCHAR environment variable 4-45R
 - creating character mapping files for 5-5R
 - data types for 1-19R
 - database server compatibility 1-19R
 - DBAPICODE environment variable 4-47R
 - DBNLS environment variable 4-48R
 - environment variables listed 4-9R
 - error messages for incompatibility 1-21R
 - functionality in Informix products 1-22R
 - functionality listed 1-15R
 - installation notes for language supplement 1-22R

LANG environment variable 4-49R
 LC_COLLATE environment variable 4-50R
 LC_CTYPE environment variable 4-51R
 LC_MONETARY environment variable 4-52R
 LC_NUMERIC environment variable 4-53R
 LC_TIME environment variable 4-53R
 NCHAR data type 3-17R
 NVARCHAR data type 3-18R
 populating with dbimport 5-18R
 setting environment variables 1-16R, 4-44R
 specifying a language environment 2-25T
 viewing characteristics of 1-18R
 NLS Database
 description of 1-16T
 NLS database
 versus non-NLS database 1-19R
 Nondecomposable attributes 8-15T
 Nonsequential access. *See* Disk access, nonsequential.
 Normal form 8-29T
 Normalization
 benefits 8-29T
 first normal form 8-29T
 of data model 8-29T
 rules 8-29T
 rules, summary 8-32T
 second normal form 8-31T
 third normal form 8-32T
 NOT CLUSTER keywords
 syntax in ALTER INDEX 1-12S
 use in ALTER TABLE 1-13S
 NOT FOUND keywords, in the
 WHENEVER statement 1-398S, 1-400S
 NOT IN keywords, use in Condition subquery 1-414S
 NOT keyword
 syntax
 in Condition segment 1-404S, 1-405S
 with BETWEEN keyword 1-327S
 with IN keyword 1-329S
 use
 in Condition segment 1-410S
 with LIKE, MATCHES keywords 1-328S
 NOT logical operator 2-36T
 NOT NULL keywords
 syntax
 in ALTER TABLE 1-17S
 in CREATE TABLE 1-88S
 use
 in ALTER TABLE 1-27S
 in CREATE TABLE 1-90S, 9-24T
 with IS keyword 1-327S
 NOT operator
 condition 1-405S
 NOT relational operator 2-32T
 NOT WAIT keywords, in SET LOCK MODE 1-370S
 NULL keyword, ambiguous as
 procedure variable 1-480S
 NULL relational operator 2-35T
 NULL value
 testing in BYTE expression 3-5R
 testing with TEXT data type 3-21R
 Null value
 checking for in SELECT 1-184S, 1-318S
 defined 8-23T
 detecting in ESQL 5-15T
 in INSERT statement 4-7T
 in SPL IF statement 2-25S
 inserting with the VALUES clause 1-252S
 restrictions in primary key 8-22T
 returned implicitly by stored
 procedure 2-38S
 specifying as default value 1-20S
 testing for 2-35T
 updating a column 1-387S
 used in Condition with NOT operator 1-405S
 used in the ORDER BY clause 1-339S
 with logical operator 2-36T
 with SPL WHILE statement 2-46S
 NUMERIC data type. *See* DECIMAL data type.
 NVARCHAR data type
 description of 3-18R, 9-15T
 syntax 1-425S
 versus VARCHAR data type 1-19R

O

- OF keyword
 - syntax in DECLARE 1-145S
 - use in DECLARE 1-153S
- OLD keyword
 - in DELETE REFERENCING clause 1-119S
 - in INSERT REFERENCING clause 1-119S
 - in UPDATE REFERENCING clause 1-120S
- ON DELETE CASCADE keyword
 - DELETE trigger event 1-112S
- ON EXCEPTION statement
 - placement of 2-32S
 - scope of control 14-29T
 - syntax 2-31S
 - trapping errors 14-27T
 - user-generated errors 14-30T
- ON keyword
 - syntax
 - in CREATE INDEX 1-63S
 - in GRANT 1-231S
 - in REVOKE 1-300S
 - use
 - in CREATE INDEX 1-64S
 - in GRANT 1-237S
- ON-Archive utility, using the ARC_DEFAULT environment variable 4-14R
- ON-Archive utility, using the ARC_KEYPAD environment variable 4-15R
- oncheck utility 10-7T, 10-12T, 13-31T
- ONCONFIG environment variable 4-39R
- onconfig file, specifying with ONCONFIG 4-40R
- One-to-many relationship 8-9T, 8-11T
- One-to-one relationship 8-9T, 8-11T
- OnLine 5-10T
- On-line
 - files Intro-8R, Intro-11S, 6T
 - help Intro-9R, Intro-11S, 7T
- onload utility 4-27T, 10-11T
- onstat utility 10-12T
- onunload utility 4-27T, 10-11T
- OPEN statement
 - activating a cursor 5-20T
 - constructing the active set 1-264S
 - opening a procedure cursor 1-265S
 - opening an insert cursor 1-266S
 - opening select or update cursors 1-264S, 5-20T
 - reopening a cursor 1-267S
 - substituting values for ? parameters 1-268S
 - syntax 1-263S
 - with concatenation operator 1-432S
 - with FREE 1-269S
- Opening a cursor 5-20T, 5-24T
- Operating system
 - DOS 12-4T
 - UNIX 12-4T
- Optical device 12-6T
- Optimization, specifying a high or low level 1-374S
- Optimizer
 - and GROUP BY 13-10T, 13-12T, 13-27T
 - and ORDER BY 13-10T, 13-12T, 13-27T
 - and SET OPTIMIZATION statement 1-374S, 13-36T
 - autoindex path 13-29T
 - composite index use 13-31T
 - data distributions 13-29T
 - disk access 13-14T
 - display query plan 13-12T
 - filter selectivity 13-30T
 - index not used by 13-32T, 13-35T
 - index used by 13-9T, 13-10T
 - methods of 13-8T
 - query plan 13-22T
 - sort merge join 13-27T
 - sorting 13-12T
 - specifying high or low level of optimization 13-36T
 - system catalog use 13-9T
 - when index not used 13-33T
 - with UPDATE STATISTICS 1-394S
- Optimizing
 - a query 1-360S
 - a server 1-374S
 - across a network 1-374S
 - techniques 13-3T
- Optional, entity in relationship 8-9T
- OR keyword
 - syntax in Condition segment 1-404S

- use in Condition segment 1-417S
- OR logical operator 2-36T
- OR relational operator 2-33T
- ORDER BY keywords
 - and NLS 2-25T
 - ascending order 1-338S, 2-14T
 - DESC keyword 2-15T, 2-25T
 - descending order 1-338S
 - display label with 2-52T
 - indexes for 13-10T, 13-34T
 - multiple columns 2-15T
 - relation to GROUP BY 3-7T
 - restrictions in INSERT 1-253S, 4-10T
 - restrictions in view 11-22T
 - restrictions with FOR UPDATE 6-8T
 - select columns by number 1-339S, 2-24T
 - sorting rows 2-13T, 13-12T
 - syntax in SELECT 1-310S
 - use
 - in SELECT 1-337S
 - with UNION operator 1-344S
- Order of execution, of action statements 1-122S
- orders table in stores6 database, columns in A-3R
- Outer join
 - forming 1-324S
- OUTER keyword, with FROM keyword in SELECT 1-323S
- Output from TRACE command 15-14T
- OUTPUT statement, syntax and use 1-271S
- Owner
 - in ALTER TABLE 1-16S
 - in CREATE SYNONYM 1-80S
 - in dbschema 5-35R
 - in Index Name segment 1-419S, 1-484S, 1-495S, 1-505S
 - in RENAME COLUMN 1-294S
 - in RENAME TABLE 1-296S
 - in Table Name segment 1-469S, 1-507S
 - in View Name segment 1-510S
 - of view in CREATE VIEW 1-511S
- Owner-privileged procedure 14-13T
- Ownership 11-7T

P

- Page buffer
 - cost of nonsequential access 13-18T
 - description of 13-16T
 - effect on performance 13-17T
 - restrictions with BLOB data 10-19T
- PAGE keyword
 - use in ALTER TABLE 1-33S
 - use in CREATE TABLE 1-108S
- Page lock 7-8T
- Page, definition of 10-4T
- Parameter
 - BYTE or TEXT in SPL 2-13S
 - in CALL statement 2-4S
 - to a stored procedure 14-25T
- Parameterizing a statement
 - with SQL identifiers 1-279S
- Parent-child relationship 1-21S, 1-94S
- Parts explosion 5-26T
- PATH environment variable 4-54R, 12-14T
- Pathname
 - including in SQLEXEC 4-42R
 - specifying with DBPATH 4-21R
 - specifying with PATH 4-54R
- Percent (%) sign, wildcard in Condition segment 1-410S
- PERFORM keyword, in the WHENEVER statement 1-398S
- Performance
 - adding indexes 13-34T
 - assigning table to dedicated disk 10-6T
 - bottleneck tables 10-35T
 - buffered log 9-22T
 - clustered index 10-25T
 - depends on concurrency 7-3T
 - disk access 13-14T, 13-17T, 13-34T
 - disk access by rowid 13-18T
 - disk arm motion 10-8T
 - disk latency 13-17T
 - dropping indexes to speed modifications 10-24T
 - duplicate keys slow modifications 10-22T
 - effect of
 - BLOB location 10-18T
 - correlated subquery 13-32T

-
- filter expression 13-23T, 13-32T, 13-33T
 - index damage 13-31T
 - indexes 10-21T to 10-22T, 13-26T
 - regular expressions 13-32T
 - table size 13-19T, 13-34T
 - updates 13-31T
 - filter selectivity 13-30T
 - “hot spots,” finding 13-7T
 - improved by specifying optimization level 13-36T
 - improved with temporary table 13-36T
 - improving 13-27T
 - increasing with stored procedures 14-4T
 - index time during modification 10-20T
 - interleaved dbspaces 10-10T
 - journal updates 10-33T
 - measurement 13-6T
 - multiple access arms per table 10-7T
 - network access 13-20T
 - nonsequential access 13-18T
 - optimizing 13-3T to 13-42T
 - references to other books 13-3T
 - reorganizing a dbspace 10-10T
 - row access 13-16T
 - seek time 13-17T
 - sequential access 13-17T, 13-34T
 - sorting replaces nonsequential access 13-37T
 - splitting tall tables 10-29T
 - splitting wide tables 10-28T
 - time costs of query 13-14T
 - use of derived data 10-30T
 - use of redundant data 10-31T
 - using CHAR instead of NCHAR 1-19R
 - Performance analysis
 - “80-20 rule” 13-7T
 - measurement 13-6T
 - methods 13-27T to 13-41T
 - nonsequential access 13-37T to 13-41T
 - optimizing techniques 13-4T
 - setting up test environment 13-28T
 - timing
 - from 4GL program 13-6T
 - from command script 13-6T
 - from watch 13-6T
 - using query plan 13-29T
 - verifying problem 13-4T
 - Permission, with SYSTEM 2-40S
 - Phantom row 1-367S
 - PIPE keyword, in the OUTPUT statement 1-272S
 - Pipe symbol. *See* concatenation operator.
 - Pipes, unnamed 12-7T
 - Plus (+) sign, arithmetic operator 1-431S
 - Populating tables 9-27T
 - POW function
 - syntax in expression 1-444S
 - use in expression 1-446S
 - Precedence, rules for environment variables 4-7R
 - PRECISION field
 - setting with SET DESCRIPTOR 1-358S
 - with GET DESCRIPTOR 1-214S
 - PREPARE statement
 - description of 5-29T
 - error return in SQLERRD 5-12T
 - executing 1-184S
 - increasing performance efficiency 1-283S
 - missing WHERE signalled 5-10T
 - multiple SQL statements 5-30T
 - multi-statement text 1-277S, 1-281S
 - parameterizing a statement 1-278S
 - parameterizing for SQL identifiers 1-279S
 - preparing GRANT 11-13T
 - question (?) mark as placeholder 1-273S
 - releasing resources with FREE 1-208S
 - restrictions with SELECT 1-276S
 - statement identifier use 1-274S
 - syntax 1-273S
 - valid statement text 1-275S
 - with concatenation operator 1-432S
 - Prepared statement
 - describing returned values with DESCRIBE 1-162S
 - executing 1-184S
 - prepared object limit 1-274S
 - valid statement text 1-275S
 - See also* PREPARE statement.
 - PREVIOUS keyword
 - syntax in FETCH 1-194S
 - use in FETCH 1-196S
 - Primary key
 - definition of 8-22T

- restrictions with 8-22T
- Primary key constraint 1-22S
 - composite 8-23T
 - data type conversion 1-28S
 - defining column as 1-93S
 - definition of 4-19T
 - dropping 1-32S
 - enforcing 1-86S
 - modifying a column with 1-28S
 - referencing 1-22S
 - requirements for 1-20S, 1-93S
 - rules of use 1-31S, 1-94S
- PRIMARY KEY keywords
 - in ALTER TABLE 1-29S
 - in ALTER TABLE statement 1-20S
 - in CREATE TABLE 1-91S, 1-92S
- Primary site 12-20T
- Printing, specifying print program with
 - DBPRINT 4-24R
- PRIOR keyword
 - syntax in FETCH 1-194S
 - use in FETCH 1-196S
- Privilege
 - Alter 1-236S, 11-8T
 - and views 11-27T to 11-30T
 - ANSI-compliant databases 1-12R
 - automating grants of 11-13T
 - column-level 11-10T
 - Connect 1-232S, 11-6T
 - DBA 1-233S, 11-7T
 - default for stored procedures 14-14T
 - default for table using CREATE
 - TABLE 1-85S
 - Delete 1-236S, 11-8T, 11-28T
 - displaying 4-16T
 - displaying with the INFO statement
 - 1-242S
 - encoded in system catalog 2-28R,
 - 2-32R, 11-9T
 - Execute 11-12T, 14-14T
 - for triggered action 1-129S
 - granting 11-5T to 11-15T
 - Index 1-236S, 11-8T
 - Insert 1-236S, 11-8T, 11-28T
 - needed
 - to create a view 1-239S, 11-27T
 - to drop an index 1-174S
 - to modify data 1-236S, 4-15T
 - on a synonym 1-80S
 - on a view 1-137S, 11-28T
 - on stored procedures 14-13T
 - overview 1-8T
 - Resource 1-233S, 11-7T, 13-34T
 - Select 11-8T, 11-10T, 11-27T
 - Update 1-236S, 11-8T, 11-10T, 11-28T
 - when privileges conflict 1-232S
 - with DBA-privileged procedures
 - 14-13T
 - with owner-privileged procedures
 - 14-13T
 - See also* Database-level privilege.
 - See also* Table-level privilege.
- PRIVILEGES FOR keywords, in INFO
 - statement 1-243S
- PRIVILEGES keyword
 - syntax
 - in GRANT 1-235S
 - in REVOKE 1-301S
 - use
 - in GRANT 1-236S
 - in REVOKE 1-302S
- Procedure cursor
 - opening 1-265S
 - reopening 1-267S
- procedure cursor 1-148S
- Procedure name
 - conflict with function name 1-495S
 - naming conventions 1-495S
- Procedure, stored. *See* Stored procedure.
- Processing, distributed 12-8T
- Projection, described 2-7T
- Project, description of 1-13T
- Promotable lock 1-152S, 7-6T, 7-9T
- PSORT_DBTEMP environment variable
 - 4-40R
- PSORT_NPROCS environment variable
 - 4-41R
- PUBLIC keyword
 - privilege granted to all users 11-6T
 - syntax
 - in GRANT 1-231S
 - in REVOKE 1-300S
 - use
 - in GRANT 1-234S
 - in REVOKE 1-302S
- PUT statement
 - constant data with 6-11T
 - count of rows inserted 6-11T
 - impact on trigger 1-112S
 - sends returned data to buffer 6-10T

- source of row values 1-285S
- syntax 1-284S
- use in transactions 1-285S
- with concatenation operator 1-432S
- with FLUSH 1-285S

Q

- Qualifier, field
 - for DATETIME 3-8R, 1-428S, 1-488S
 - for INTERVAL 3-13R, 1-485S, 1-491S

- Query
 - cyclic 4-22T
 - improving performance of 13-27T to 13-41T
 - performance of 13-3T to 13-42T
 - pipng results to another program 1-272S
 - self-referencing 4-22T
 - sending results to an operating system file 1-271S
 - sending results to another program 1-272S
 - stated in terms of data model 1-7T
 - time costs of 13-14T

- Query optimization information statements 1-6S

- Query optimizer. *See* Optimizer.

- Query plan
 - autoindex path 13-29T
 - chosen by optimizer 13-11T
 - description of 13-22T
 - display with SET EXPLAIN 13-12T
 - indexes in 13-26T
 - use in analyzing performance 13-29T

- Question (?) mark
 - as placeholder in PREPARE 1-273S
 - naming variables in PUT 1-287S
 - replacing with USING keyword 1-268S
 - wildcard in Condition segment 1-411S

- Quoted string
 - in expression 1-436S
 - syntax
 - in Condition segment 1-405S
 - in expression 1-462S
 - in INSERT 1-250S
 - use
 - in expression 1-438S
 - in INSERT 1-498S

- with LIKE, MATCHES keywords 1-328S

- See also* Quoted String segment.

- Quoted String segment
 - DATETIME, INTERVAL values as strings 1-498S

- syntax 1-497S

- wildcards 1-498S

- with LIKE in a condition 1-498S

- Quotes, single and double Intro-6S

R

- RAISE EXCEPTION statement
 - exiting a loop 14-23T
 - syntax 2-36S

- REAL data type. *See* SMALLFLOAT data type.

- RECOVER TABLE statement
 - archiving a database with audit trails 1-292S
 - manipulating audit trail file 1-293S
 - syntax 1-292S

- Recursion, in a stored procedure 14-24T

- Recursive relationship 8-11T, 8-28T

- Redundant data, introduced for performance 10-31T

- Redundant relationship 8-29T

- REFERENCES FOR keywords, in INFO statement 1-243S

- REFERENCES keyword
 - in ALTER TABLE 1-21S
 - in CREATE TABLE 1-94S, 1-96S
 - syntax

- in GRANT 1-235S

- in REVOKE 1-301S

- use

- in GRANT 1-236S

- in REVOKE 1-302S

- References privilege
 - definition of 1-236S
 - displaying with the INFO statement 1-243S

- REFERENCING clause
 - DELETE REFERENCING clause 1-119S

- INSERT REFERENCING clause 1-118S

- UPDATE REFERENCING clause 1-120S

- using referencing 1-125S, 15-9T
- Referential constraint
 - and a DELETE trigger 1-112S
 - data type restrictions 1-96S
 - definition of 1-21S, 1-94S, 4-19T
 - dropping 1-32S
 - enforcing 1-86S
 - modifying a column with 1-28S
 - rules of use 1-94S
- Referential integrity 4-19T
 - defining primary and foreign keys 8-24T
- Regular expression
 - effect on performance 13-32T
 - evaluating with NLS 1-15R
- Relational calculus. *See also* relational model.
- Relational database, defined 1-11T
- Relational model
 - attribute 8-14T
 - denormalizing 10-26T
 - description of 1-11T, 8-3T to 8-33T
 - entity 8-4T
 - join 2-8T
 - many-to-many relationship 8-11T
 - normalizing data 8-29T
 - one-to-many relationship 8-11T
 - one-to-one relationship 8-11T
 - projection 2-6T
 - resolving relationships 8-26T
 - rules for defining tables, rows, and columns 8-20T
 - selection 2-5T
- Relational operation 2-5T
- Relational operator
 - BETWEEN 2-32T
 - equals 2-30T
 - EXISTS 3-33T
 - IN 3-33T
 - in a WHERE clause 2-29T to 2-45T
 - in Condition segment 1-405S
 - LIKE 2-37T
 - NOT 2-32T
 - NULL 2-35T
 - OR 2-33T
 - segment 1-500S
 - with WHERE keyword in SELECT 1-326S
- Relationship
 - attribute 8-15T
 - cardinality 8-9T, 8-13T
 - complex 8-28T
 - connectivity 8-8T, 8-11T
 - defining in data model 8-8T
 - entity 8-5T
 - existence dependency 8-9T
 - mandatory 8-9T
 - many-to-many 8-9T, 8-11T
 - many-to-many, resolving 8-26T
 - one-to-many 8-9T, 8-11T
 - one-to-one 8-9T, 8-11T
 - optional 8-9T
 - recursive 8-28T
 - redundant 8-29T
 - using matrix to discover 8-10T
- RELATIVE keyword
 - syntax in FETCH 1-194S
 - use in FETCH 1-196S
- Relay Module
 - SQLRM environment variable 4-42R
 - SQLRMDIR environment variable 4-43R
- Release notes Intro-9R, Intro-11S, 6T
- Remote database server 12-8T
- RENAME COLUMN statement
 - restrictions 1-294S
 - syntax 1-294S
- RENAME TABLE statement
 - ANSI-compliant naming 1-296S
 - syntax 1-296S
- REPAIR TABLE statement, syntax and use 1-299S
- Repeatable Read isolation level
 - description of 1-367S, 7-12T
 - emulating during update 1-202S
- REPEATABLE READ keywords, syntax in SET ISOLATION 1-366S
- Replication of data 12-19T
- Report generator 1-17T
- Reserved words 1-470S
- Resolution
 - in UPDATE STATISTICS 1-396S, 1-397S
 - with data distributions 1-396S
- RESOURCE keyword
 - use in GRANT 1-233S
 - use in REVOKE 1-303S
- Resource manager 12-11T
- Resource privilege 1-233S, 11-7T, 13-34T

-
- Restricting access, using file system 11-4T
 - Result of triggering statement 1-123S
 - RETURN statement
 - exiting a loop 14-23T
 - returning insufficient values 2-38S
 - returning null values 2-38S
 - syntax 2-38S
 - REVOKE statement
 - column-specific privileges 1-303S
 - database-level privileges 1-303S
 - granting privileges 11-5T to 11-15T
 - in embedded SQL 5-34T to 5-37T
 - privileges needed 1-301S
 - syntax 1-300S
 - table-level privileges 1-301S
 - with a view 11-29T
 - ROLLBACK WORK statement
 - cancels a transaction 4-25T
 - closes cursors 7-18T
 - releases locks 7-8T, 7-18T
 - sets SQLCODE 6-5T
 - syntax 1-306S
 - use with WHENEVER 1-36S, 1-42S, 1-307S
 - with DROP DATABASE 1-173S
 - with DROP INDEX statement 1-175S
 - with DROP PROCEDURE statement 1-176S
 - with DROP SYNONYM statement 1-177S
 - with DROP TABLE statement 1-179S
 - with DROP TRIGGER statement 1-181S
 - with DROP VIEW statement 1-183S
 - ROLLFORWARD DATABASE statement
 - applies log to restored database 4-26T
 - exclusive locking 1-308S
 - syntax 1-308S
 - Root dbspace, definition of 10-5T
 - ROOT function
 - syntax in expression 1-444S
 - use in expression 1-446S
 - ROUND function
 - syntax in expression 1-444S
 - use in expression 1-446S
 - Row
 - cost of reading from disk 13-16T
 - defined 2-5T
 - defining 8-20T
 - deleting 1-159S, 4-4T
 - description of 1-12T
 - engine response to locked row 1-370S
 - in relational model 1-12T, 8-20T
 - inserting 4-6T
 - inserting through a view 1-246S
 - inserting with a cursor 1-248S
 - multi-row queries with FETCH 1-195S
 - order, guaranteeing independence of 1-117S
 - phantom 1-367S
 - retrieving with FETCH 1-197S
 - rowid definition 1-197S
 - size of fixed-length 10-13T
 - updating through a view 1-384S
 - writing buffered rows with FLUSH 1-204S
 - ROW keyword
 - use in ALTER TABLE 1-33S
 - use in CREATE TABLE 1-108S
 - Row lock 7-8T
 - Rowid
 - locating internal row numbers 3-17T
 - use in a column expression 1-435S
 - use in a join 3-16T
 - use in ORDER BY clause 1-338S
 - used as column name 1-474S
 - Rowid function 13-18T
 - ROWID keyword 1-435S
 - Rules for stored procedures 1-128S
 - Run-time program
 - setting DBANSIWARN 4-15R
 - setting INFORMIXCOBDIR 4-32R
- ## S
- SCALE field
 - setting with SET DESCRIPTOR 1-358S
 - with GET DESCRIPTOR 1-214S
 - Schema. *See* Data Model.
 - Scroll cursor
 - active set 5-25T
 - definition of 1-149S, 5-23T
 - use of 1-149S
 - with FETCH 1-196S
 - with hold, in a transaction 1-369S
 - SCROLL keyword
 - syntax in DECLARE 1-145S

- use in DECLARE 1-149S, 5-23T
- Search condition. *See* Condition segment.
- secheck utility 13-31T
- SECOND keyword
 - syntax
 - in DATETIME data type 1-428S
 - in INTERVAL data type 1-485S
 - use
 - as DATETIME field qualifier 3-8R, 1-488S
 - as INTERVAL field qualifier 3-14R, 1-491S
- Second normal form 8-31T
- Secondary site 12-20T
- Security
 - constraining inserted values 11-20T, 11-25T
 - database-level privileges 11-5T
 - making database inaccessible 11-5T
 - restricting access to columns 11-20T
 - restricting access to rows 11-20T, 11-21T
 - restricting access to view 11-27T
 - table-level privileges 11-10T
 - using host file system 11-4T
 - using operating system facilities 11-4T
 - with stored procedures 11-3T
- Seek time 13-17T
- Select
 - description of 1-13T
- Select cursor
 - definition of 1-148S
 - opening 1-264S, 5-20T
 - reopening 1-267S
 - use of 1-148S, 5-20T
- SELECT keyword
 - ambiguous use as procedure variable 1-480S
 - syntax
 - in GRANT 1-235S
 - in REVOKE 1-301S
 - use
 - in GRANT 1-236S
 - in REVOKE 1-302S
- Select list
 - display label 2-49T
 - expressions in 2-46T
 - functions in 2-53T to 2-66T
 - labels in 3-48T
 - selecting all columns 2-12T
 - selecting specific columns 2-18T
 - specifying a substring in 2-27T
- Select privilege
 - column level 11-10T
 - definition of 1-236S, 11-8T
 - with a view 11-27T
- SELECT statement
 - active set 2-29T
 - aggregate functions in 1-462S, 2-53T
 - alias names 2-79T
 - and COLLCHAR environment variable 4-46R
 - and LC_COLLATE environment variable 4-51R
 - and NLS collation order 1-15R
 - as an argument to a stored procedure 2-4S
 - assigning values with 14-21T
 - associating with cursor with DECLARE 1-148S
 - BETWEEN condition 1-327S
 - column numbers 1-339S
 - compound query 3-43T
 - cursor for 5-19T, 5-20T
 - date-oriented functions in 2-56T
 - describing returned values with DESCRIBE 1-162S
 - description of advanced 3-4T to 3-55T
 - description of simple 2-3T to 2-83T
 - display label 2-49T
 - DISTINCT keyword 2-19T
 - embedded 5-13T to 5-16T
 - for joined tables 2-69T to 2-83T
 - for single tables 2-11T to 2-66T
 - FROM Clause 1-323S
 - functions 2-53T to 2-66T
 - GROUP BY clause 1-334S, 3-4T
 - HAVING clause 1-336S, 3-9T
 - IN condition 1-327S
 - in FOR EACH ROW section 1-117S
 - in modifiable view 11-24T
 - INTO clause with ESQL 1-318S
 - INTO TEMP clause 1-341S, 2-83T
 - IS NULL condition 1-327S
 - join 2-71T to 2-79T
 - joining tables in WHERE clause 1-331S
 - LIKE or MATCHES condition 1-328S
 - multiple-table 2-69T
 - natural join 2-75T

null values in the ORDER BY clause
 1-339S
 ORDER BY clause 1-337S, 2-13T
 ORDER BY clause and NLS 1-15R
 outer join 3-21T to 3-31T
 privilege for 11-6T, 11-8T
 relational-operator condition 1-326S
 restrictions with INTO clause 1-276S
 rowid 3-16T, 3-21T
 ROWID keyword 1-435S
 SELECT clause 1-312S, 2-12T to 2-28T
 select numbers 1-339S
 selecting a substring 2-27T
 selecting expressions 2-46T
 selection list 2-12T
 self-join 3-11T
 single-table 2-11T
 singleton 1-318S, 2-29T
 subquery 3-32T to 3-42T
 subquery with WHERE keyword
 1-326S
 syntax 1-310S
 UNION operator 1-344S, 3-43T
 use of expressions 1-313S
 using
 for join 2-8T
 for projection 2-7T
 for selection 2-5T
 WHERE clause and NLS 1-15R
 with
 Condition segment 1-404S
 DECLARE 1-145S
 FOREACH 2-20S
 INSERT 1-253S
 INTO keyword 1-198S
 LET 2-29S
 writing rows retrieved to an ASCII file
 1-378S
 Selection, described 2-5T
 Self-join
 assigning column names with INTO
 TEMP 3-12T
 description of 1-333S, 3-11T
 See also Join.
 Self-referencing query 3-11T, 4-22T
 Self-referencing query. *See also* Self-join.
 Semantic integrity 4-18T, 9-3T
 Sequential access. *See* Disk access,
 sequential.
 Sequential cursor
 definition of 1-149S, 5-23T
 use of 1-149S
 with FETCH 1-195S
 SERIAL data type
 description of 3-19R, 9-7T
 generated number in SQLERRD 5-12T
 in ALTER TABLE 1-17S
 in INSERT 1-251S
 inserting a starting value 4-8T
 inserting values 3-19R
 resetting values 3-19R
 syntax 1-425S
 treatment by dbschema 5-34R
 with stored procedures 2-8S
 Server. *See* Database server.
 SET clause 1-125S, 4-14T
 SET CONNECTION statement
 syntax and use 1-346S
 with concatenation operator 1-432S
 SET CONSTRAINTS statement
 syntax and use 1-349S
 use with CREATE TRIGGER 1-131S
 SET DEBUG FILE TO statement
 syntax and use 1-351S
 with TRACE 2-42S
 SET DESCRIPTOR statement
 syntax 1-353S
 the VALUE option 1-355S
 with concatenation operator 1-432S
 X/Open mode 1-357S
 Set difference 3-53T
 SET EXPLAIN statement
 interpreting output 1-361S, 13-29T
 MERGE JOIN information 1-362S
 optimizer access paths 1-361S
 output examples 1-362S
 SORT SCAN information 1-362S
 syntax 1-360S
 writes query plan 13-12T
 Set intersection 3-51T
 SET ISOLATION statement
 controlling the effect of locks 4-28T
 default database levels 1-368S
 definition of isolation levels 1-367S
 effects of isolation 1-368S
 restrictions 7-9T
 syntax 1-366S
 SET keyword
 syntax in UPDATE 1-383S
 use in UPDATE 1-386S, 4-12T

SET LOCK MODE statement
 controlling the effect of locks 4-28T
 description of 7-13T
 kernel locking 1-370S
 setting wait period 1-371S
 syntax 1-370S
 SET LOG statement
 buffered vs. unbuffered 1-372S, 9-22T
 syntax 1-372S
 SET OPTIMIZATION statement, syntax
 and use 1-374S
 Setting environment variables 4-5R
 SHARE keyword, syntax in LOCK
 TABLE 1-260S
 Shared memory
 network connection 12-7T
 Shared memory parameters, specifying
 file with ONCONFIG 4-39R
 Shell
 setting environment variables in a file
 4-4R
 specifying with DBREMOTECMD
 4-25R
 Simple assignment 2-28S
 SIN function
 syntax in expression 1-458S
 use in expression 1-459S
 Single-precision floating-point number,
 storage of 3-12R
 Singleton SELECT statement 1-318S,
 2-29T
 Site
 network 12-4T
 primary 12-20T
 secondary 12-20T
 SITENAME function
 returns servername 1-438S
 syntax
 in expression 1-436S
 in INSERT 1-250S
 use
 in ALTER TABLE 1-18S
 in CREATE TABLE 1-88S
 in expression 1-438S
 in INSERT 1-252S
 in SELECT 2-63T, 2-65T, 3-20T
 Slash (/), arithmetic operator 1-431S
 SMALLFLOAT data type
 changing data types 3-23R
 description of 3-20R, 9-9T
 syntax 1-425S
 SMALLINT data type
 changing data types 3-23R
 description of 3-20R, 9-7T
 syntax 1-425S
 using as default value 1-19S, 1-89S
 SOME keyword
 beginning a subquery 1-330S, 3-33T
 use in Condition subquery 1-415S
 Sort merge join 13-27T
 Sorting
 avoiding nonsequential access 13-37T
 avoiding with temporary table 13-36T
 effect on performance 13-33T
 in a combined query 1-344S
 in SELECT 1-337S
 nested 2-15T
 optimizer estimates cost 13-12T
 PSORT_DBTEMP environment
 variable 4-40R
 PSORT_NPROCS environment
 variable 4-41R
 sort merge join 13-27T
 time costs of 13-15T
 when there are multiple active NLS
 locales 1-20R
 with NLS activated 1-15R, 2-25T
 with ORDER BY 2-14T
 See also ORDER BY keywords.
 Space ()
 as delimiter in DATETIME 3-9R
 as delimiter in INTERVAL 3-15R
 Specifying ANSI-compliance 1-11R
 SPL
 flow control statements 14-22T
 program variable 5-5T
 relation to SQL 14-3T
 See also Stored procedure.
 SQL
 ANSI standard 1-15T
 cursor 5-19T
 description of 1-14T
 error handling 5-16T
 history 1-14T
 in NLS-ready products 1-22R
 Informix SQL and ANSI SQL 1-15T
 interactive use 1-17T
 optimizing. *See* Optimizer.
 standardization 1-14T

statement types 1-5S

SQL Communications Area (SQLCA)
 altered by end of transaction 6-5T
 description of 5-8T
 effect of setting DBANSIWARN 4-15R
 inserting rows 6-11T
 result after CLOSE 1-39S
 result after DATABASE 1-140S
 result after DESCRIBE 1-163S
 result after FETCH 1-202S
 result after FLUSH 1-204S
 result after OPEN 1-264S, 1-265S
 result after PUT 1-290S
 result after SELECT 1-321S
 returning NLS error messages to
 1-19R
See also SQLAWARN, SQLCODE, and
 SQLERRD.

SQL DESCRIPTOR clause. *See*
 DESCRIBE statement.

SQL Descriptor. *See* SQL
 Communications Area (SQLCA).

SQL statements
 FILE 5-23R
 INSERT 5-23R

SQLAWARN array
 description of 5-12T
 syntax of naming 5-11T
 with PREPARE 5-30T

sqlca record and EXECUTE statement
 1-186S

SQLCA. *See* SQL Communications Area.

SQLCODE field
 after opening cursor 5-20T
 description of 5-11T
 end of data on SELECT only 6-14T
 end of data signalled 5-17T
 set by DELETE 6-4T
 set by DESCRIBE 5-33T
 set by PUT, FLUSH 6-11T

sqlda structure
 syntax
 in DESCRIBE 1-162S
 in EXECUTE 1-184S
 in FETCH 1-194S
 in OPEN 1-263S
 in PUT 1-284S
 use
 in DESCRIBE 1-164S
 in FETCH 1-201S
 in OPEN 1-269S
 in PUT 1-288S
 use with EXECUTE statement 1-186S

SQLERRD array
 count of deleted rows 6-4T
 count of inserted rows 6-11T
 count of rows 6-14T
 description of 5-12T
 syntax of naming 5-11T

SQLERROR keyword, in the
 WHENEVER statement 1-398S

SQLEXEC environment variable 4-41R

sqlxecd 12-14T

sqlhosts. *See*
 SIFORMIXDIR/etc/sqlhosts.

SQLNOTFOUND
 error conditions with EXECUTE
 statement 1-188S

SQLRM environment variable 4-42R

SQLRMDIR environment variable 4-43R

SQLSTATE
 in databases that are not
 ANSI-compliant 5-18T
 use with a cursor 5-21T

SQLWARNING keyword, in the
 WHENEVER statement 1-400S

SQRT function
 syntax in expression 1-444S
 use in expression 1-447S

START DATABASE statement
 adding a transaction log 9-24T
 syntax and use 1-376S

state table in stores6 database, columns
 in A-6R

Statement
 naming with NLS 1-15R
 SQL, ANSI compliance and
 DBANSIWARN 4-15R
 SQL, CONNECT and
 INFORMIXSERVER 4-37R
 SQL, CREATE TABLE and
 COLLCHAR 4-46R
 SQL, DESCRIBE and COLLCHAR
 4-46R
 SQL, editing and DBEDIT 4-18R
 SQL, LOAD and DBDELIMITER
 4-18R
 SQL, SELECT and COLLCHAR 4-46R

SQL, SELECT and LC_COLLATE 4-51R
 SQL, UNLOAD and DBDELIMITER 4-18R
 SQL, UPDATE STATISTICS and DBUPSPACE 4-30R
 Statement identifier
 associating with cursor 1-148S
 definition of 1-274S
 releasing 1-274S
 syntax
 in DECLARE 1-145S
 in DESCRIBE 1-162S
 in EXECUTE 1-184S
 in FREE 1-207S
 in PREPARE 1-273S
 use
 in DECLARE 1-154S
 in FREE 1-208S
 in PREPARE 1-274S
 Statement types 1-5S
 Statement variable name, definition 1-190S
 Static SQL 5-5T
 STATUS FOR keywords, in INFO statement 1-244S
 STATUS variable (4GL) 5-11T
 Status, displaying with INFO statement 1-244S
 stock table in stores6 database, columns in A-4R
 STOP keyword, in the WHENEVER statement 1-398S, 1-402S
 Storage device 12-6T
 Stored procedure
 altering 14-13T
 as triggered action 1-128S, 15-10T
 branching 14-22T
 BYTE and TEXT data types 2-9S, 2-13S
 checking references 1-129S
 comments in 14-6T
 creating from an embedded language 14-5T
 creating from DB-Access 14-5T
 cursors with 2-20S
 DBA-privileged, use with triggers 1-129S, 14-13T
 debugging 2-42S, 14-11T
 default privileges 14-14T
 DEFINE statement 14-17T
 definition of 14-3T
 displaying contents 14-9T
 displaying documentation 14-9T
 executing 14-9T
 general programming 1-17T
 granting privileges on 1-237S, 11-12T, 14-15T
 handling multiple rows 2-39S
 header 2-8S, 14-25T
 in SELECT statements 1-315S, 2-67T
 in WHEN condition 1-122S
 introduction to 14-3T
 looping 14-23T
 name confusion with SQL functions 14-21T
 naming output file for TRACE statement 1-351S
 owner-privileged 1-129S, 14-13T
 privileges 1-129S
 privileges necessary at execution 14-14T
 program flow control 14-22T
 receiving data from SELECT 1-318S
 recursion 14-24T
 REFERENCES clause 14-18T
 returning values 14-25T
 revoking privileges on 1-301S, 14-16T
 security purposes 11-3T
 simulating errors 2-36S
 tracing triggered actions 15-12T
 use 14-3T
 variable 14-16T
 Stored Procedure Language. *See* SPL.
 stores6 database
 call_type table columns A-5R
 catalog table columns A-4R
 copying Intro-10R, Intro-13S, 8T
 creating Intro-10R, Intro-13S, 8T
 customer table columns A-2R
 cust_calls table columns A-5R
 data values A-15R
 description of A-1R
 items table columns A-3R
 manufact table columns A-5R
 map of A-6R
 orders table columns A-3R
 overview Intro-9R, Intro-12S, 7T
 primary-foreign key relationships A-8R to A-15R
 state table columns A-6R
 stock table columns A-4R
 structure of tables A-2R

Structured Query Language. *See* SQL.

Subquery

- beginning with ALL/ANY/SOME keywords 1-330S
- beginning with EXISTS keyword 1-330S
- beginning with IN keyword 1-329S
- correlated 1-413S, 3-32T, 4-22T, 13-32T
- definition of 1-326S
- in Condition segment 1-413S
- in DELETE statement 4-6T
- in SELECT 3-32T to 3-42T
- in UPDATE-SET 4-13T
- in UPDATE-WHERE 4-12T
- performance of 13-32T
- restrictions with UNION operator 1-344S
- with DISTINCT keyword 1-313S

See also Condition segment.

Subscripting

- in a WHERE clause 2-44T
- on character columns 1-434S
- SPL variables 14-18T

Subservient table 3-21T

Substring 2-27T, 14-18T

SUM function

- as aggregate function 2-53T
- syntax in expression 1-462S
- use in expression 1-465S

Symbol table 10-27T

Synonym

- ANSI-compliant naming 1-80S
- chaining 1-83S
- chains 12-18T
- creating with CREATE SYNONYM 1-80S
- difference from alias 1-80S
- dropping 1-177S
- in ANSI-compliant database 1-14R

synonym behavior

- in ANSI-compliant database 1-14R

Synonyms for table names 12-17T

Syntax diagram

- conventions Intro-5S
- elements of Intro-9S

sysdepend system catalog table 1-183S

syssyntable 12-18T

System catalog

- accessing 2-8R
- altering contents 2-9R
- character column data type when NLS activated 1-19R
- database entries 1-57S
- description of 2-3R
- map of tables 2-33R
- NCHAR columns in 1-19R
- privileges in 4-16T, 11-9T
- querying 4-16T
- sysblobs 2-11R
- syschecks 2-11R
- syscolauth 2-12R, 1-302S, 11-9T
- syscoldepend 2-13R
- syscolumns 2-13R
- sysconstraints 2-16R
- sysdefaults 2-17R
- sysdepend 2-18R
- sysdistrib 2-18R
- sysindexes 2-20R
- sysopclstr 2-22R
- sysprocauth 2-23R
- sysprocbody 2-24R, 14-8T
- sysprocedures 2-25R
- sysprocplan 2-26R
- sysreferences 2-26R
- syssynonyms 2-27R
- syssyntable 2-27R
- systabauth 2-28R, 1-239S, 1-302S, 4-16T, 11-9T
- systables 2-29R
- systrigbody 2-31R
- systriggers 2-32R
- sysusers 2-32R, 11-9T
- sysviews 2-33R
- updating 2-9R
- updating statistics 2-9R
- used by optimizer 13-9T

System catalog tables. *See* System catalog.

System descriptor area 1-164S

- assigning values to 1-354S
- modifying contents 1-354S
- resizing 1-355S
- use with EXECUTE statement 1-187S

System name, in database name 1-422S

SYSTEM statement

- syntax 2-40S

T

Table

- adding a constraint 1-29S
 - alias in SELECT 1-323S
 - ANSI-compliant naming 1-507S
 - bottleneck 10-35T
 - candidate keys, defined 8-23T
 - changing the data type of a column 3-23R
 - checking with the CHECK TABLE statement 1-37S
 - composite key, defined 8-23T
 - contained in one dbspace 10-5T
 - creating
 - a synonym for 1-80S
 - a table 1-84S, 9-24T
 - a temporary table 1-100S
 - dedicated device for 10-6T
 - description of 1-11T
 - dropping
 - a constraint 1-32S
 - a synonym 1-177S
 - a table 1-179S
 - engine response to locked table 1-370S
 - extent sizes of 10-8T
 - fixed-length rows 10-13T
 - in mirrored storage 10-6T
 - in relational model 1-11T, 8-20T
 - interleaved dbspaces 10-10T
 - joins in Condition segment 1-331S
 - loading data with the LOAD statement 1-255S
 - lock 7-7T
 - locking
 - changing mode 1-33S
 - with ALTER INDEX 1-13S
 - with LOCK TABLE 1-260S
 - logging 1-104S
 - multiple access arms for 10-7T
 - names 12-16T
 - names, synonyms 12-17T
 - naming conventions 1-84S, 1-470S, 1-507S
 - naming with NLS 1-15R
 - optimizing queries 1-394S
 - ownership 11-7T
 - primary key in 8-22T
 - primary key, defined 8-22T
 - relation to dbspace 10-6T
 - repairing with REPAIR TABLE statement 1-299S
 - represents an entity 8-22T
 - restoring with audit trail 1-292S
 - structure in stores6 database A-2R
 - system catalog tables 2-11R to 2-33R
 - unlocking 1-381S
 - variable-length rows 10-14T
 - See also* ALTER TABLE statement.
 - See also* CREATE TABLE statement.
 - See also* Join.
 - See also* Table Name segment.
- TABLE keyword, syntax in UPDATE STATISTICS 1-393S
- Table Name segment 1-506S
- Table size
 - calculating 10-12T, 10-19T
 - cost of access 13-19T, 13-34T
 - with fixed-length rows 10-13T
 - with variable-length rows 10-14T
- Table-level privilege
 - column-specific privileges 1-303S, 11-10T
 - default with GRANT 1-237S
 - definition and use 1-236S, 11-7T
 - granting 1-235S
 - passing grant ability 1-237S
 - revoking 1-301S
 - See also* ALTER TABLE statement.
- TABLES keyword, in INFO statement 1-242S
- tabtype 2-29R, 2-30R, 1-60S
- TAN function
 - syntax in expression 1-458S
 - use in expression 1-460S
- tblspace
 - description of 10-7T
 - used for BLOB data 10-18T
- TEMP keyword
 - syntax in SELECT 1-310S
 - use in SELECT 1-341S
- TEMP TABLE keywords, syntax in CREATE TABLE 1-84S
- Temporary
 - files, specifying directory with DBTEMP 4-27R
 - tables, specifying dbspace with DBSPACETEMP 4-26R
- Temporary table
 - and active set of cursor 5-24T
 - assigning column names 3-12T
 - building distributions 1-397S

- creating constraints for 1-102S
 - DBSPACETEMP environment variable 1-100S
 - example 4-11T
 - explicit 1-100S
 - implicit 1-100S
 - location of 1-100S
 - naming 1-100S
 - shared disk space for 10-6T
 - updating statistics 1-394S
 - using to speed query 13-36T
 - when deleted 1-100S
- TERM environment variable 4-55R
- TERMCAP environment variable 4-56R, 12-14T
- termcap file
 - and TERMCAP environment variable 4-56R
 - selecting with INFORMIXTERM 4-39R
- Terminal handling
 - and TERM environment variable 4-55R
 - and TERMCAP environment variable 4-56R
 - and TERMINFO environment variable 4-56R
- terminfo directory
 - and TERMINFO environment variable 4-56R
 - selecting with INFORMIXTERM 4-39R
- TERMINFO environment variable 4-56R
- TEXT data type
 - choosing location for 10-18T
 - description of 3-21R, 9-17T
 - disk storage for 10-4T
 - estimating disk space for 10-17T
 - inserting values 3-21R
 - requirements for LOAD statement 1-257S
 - restrictions
 - with aggregate functions 3-21R
 - with GROUP BY 3-21R, 3-7T
 - with IN clause 3-21R
 - with LIKE or MATCHES 3-21R, 2-37T
 - with ORDER BY 3-21R
 - with relational expression 2-29T
 - selecting a column 3-22R
 - syntax 1-425S
 - use in Boolean expression 3-21R
 - used for performance 10-27T
 - with control characters 3-21R
 - with LENGTH function 2-64T
 - with stored procedures 2-9S, 2-13S
- Text editor, specifying with DBEDIT 4-18R
- TEXT value, displaying 2-11T
- Third normal form 8-32T
- Time function
 - restrictions with GROUP BY 1-334S
 - use in SELECT 1-314S
- Time, representing with NLS 1-15R
- TO CLUSTER keywords, in ALTER INDEX 1-12S
- TO keyword
 - in expression 1-462S
 - in GRANT 1-231S
- TODAY function
 - syntax
 - in Condition segment 1-405S
 - in expression 1-436S
 - in INSERT 1-250S
 - use
 - in ALTER TABLE 1-18S
 - in constant expression 1-439S, 2-63T, 4-8T
 - in CREATE TABLE 1-88S
 - in INSERT 1-252S
- TP/XA. *See* Transaction manager.
- TRACE command
 - output from 15-14T
- TRACE statement
 - debugging a stored procedure 14-11T
 - syntax 2-42S
- Transaction
 - and CREATE DATABASE 1-60S
 - ANSI-compliant database, effects 1-11R
 - committing with COMMIT WORK 1-43S
 - cursors closed at end 7-18T
 - description of 4-22T
 - example with DELETE 6-5T
 - global 12-21T
 - locks held to end of 7-9T
 - locks released at end 7-8T, 7-18T
 - logging 1-376S
 - recovering transactions 1-308S

- rolling back 1-4R, 1-306S
- scroll cursor and data consistency 1-369S
- starting with BEGIN WORK 1-35S
- stopping logging 1-376S
- transaction log 4-24T, 4-26T
- transaction log required 9-22T
- use signalled in SQLAWARN 5-12T
- using cursors in 1-155S
- Transaction logging
 - ANSI-compliant database, effects 1-12R
 - buffered 9-22T
 - effect on database server type 1-5R
 - establishing with CREATE DATABASE 9-21T
 - OnLine methods of 9-22T
 - renaming log 1-377S
 - stopping 1-377S
 - turning off for faster loading 9-28T
 - turning off not possible 9-24T
- Transaction log, contents of 4-25T
- Transaction manager 12-11T
- Transfer of database files 5-19R
- Transitive dependency 8-32T
- Trigger
 - creating 15-4T
 - definition of 15-3T
 - in client/server environment 1-133S
 - number on a table 1-112S
 - preventing overriding 1-132S
 - when to use 15-3T
- Trigger event
 - definition of 1-111S, 15-5T
 - example of 15-5T
 - in CREATE TRIGGER statement 1-111S
 - INSERT 1-119S
 - privileges on 1-113S
 - with cursor statement 1-112S
- Trigger name
 - assigning 15-5T
 - syntax 1-113S
- Triggered action
 - action on triggering table 1-127S
 - anyone can use 1-129S
 - BEFORE and AFTER 15-7T
 - cascading 1-118S
 - clause 1-121S
 - action statements 1-122S
 - syntax 1-121S
 - WHEN condition 1-121S
 - correlation name in 1-125S, 1-128S
 - FOR EACH ROW 15-8T
 - generating an error message 15-14T
 - in client/server environment 1-133S
 - in relation to triggering statement 15-6T
 - list
 - AFTER 1-117S
 - BEFORE 1-116S
 - FOR EACH ROW 1-116S
 - for multiple triggers 1-117S
 - sequence of 1-116S
 - merged 1-117S
 - preventing overriding 1-132S
 - statements 15-3T
 - tracing 15-12T
 - using 15-7T
 - using stored procedures 15-10T
 - WHEN condition 1-121S, 15-10T
- Triggering statement
 - affecting multiple rows 1-117S
 - execution of 1-113S
 - guaranteeing same result 1-112S
 - result of 1-123S
 - UPDATE 1-115S
- Triggering table
 - action on 1-127S
 - and cascading triggers 1-131S
- Trigonometric function
 - ACOS function 1-460S
 - ASIN function 1-460S
 - ATAN function 1-460S
 - ATAN2 function 1-461S
 - COS function 1-459S
 - SIN function 1-459S
 - TAN function 1-460S
- TRUNC function
 - syntax in expression 1-444S
 - use in expression 1-447S
- Truncation, signalled in SQLAWARN 5-12T
- Two-phase commit 12-20T
- TYPE field
 - changing from BYTE or TEXT 1-359S
 - setting in SET DESCRIPTOR 1-356S
 - setting in X/Open programs 1-357S
 - with X/Open programs 1-213S

Typographical conventions Intro-5R,
Intro-5S, 5T

U

Unbuffered logging 9-22T

Underscore (`_`), wildcard in Condition
segment 1-410S

UNION operator
description of 3-43T
display labels with 3-48T
restrictions in view 11-22T
restrictions on use 1-344S
syntax in SELECT 1-310S
use in SELECT 1-344S

Unique constraint
dropping 1-32S
modifying a column with 1-28S
rules of use 1-30S, 1-92S, 1-94S

UNIQUE keyword
constraint in CREATE TABLE 9-24T
restrictions in modifiable view 11-24T
syntax
in CREATE INDEX 1-63S
in CREATE TABLE 1-91S
in SELECT 1-312S
use
in ALTER TABLE 1-29S
in CREATE INDEX 1-64S
in CREATE TABLE 1-92S
in expression 1-462S
in SELECT 1-313S, 2-19T
no effect in subquery 1-414S

UNITS keyword
syntax in expression 1-436S
use in expression 1-442S

UNIX
BSD
default print capability 4-7R, 4-24R
viewing environment settings 4-5R
environment variables listed 4-9R
PATH environment variable 4-54R
specifying directories for
intermediate writes 4-40R
System V
default print capability 4-7R, 4-24R
terminfo library support 4-39R
viewing environment settings 4-5R
TERM environment variable 4-55R
TERMCAP environment variable
4-56R

TERMINFO environment variable
4-56R

UNIX operating system 12-4T, 12-6T

UNLOAD statement
DELIMITER clause 1-380S
exporting data to a file 9-27T
specifying field delimiter with
DBDELIMITER 4-18R
syntax 1-378S
UNLOAD TO file 1-378S
unloading VARCHAR, TEXT, or
BYTE columns 1-379S

UNLOAD TO file 1-378S

Unloading a database 5-8R

UNLOCK TABLE statement, syntax and
use 1-381S

Unnamed pipes 12-7T

Updatable view 1-139S

UPDATE clause, syntax 1-114S

Update cursor 1-148S
definition of 1-148S, 6-15T
locking considerations 1-152S
opening 1-264S
restricted statements 1-152S
use in UPDATE 1-390S
using 1-152S

Update journal 10-33T

UPDATE keyword
syntax
in GRANT 1-235S
in REVOKE 1-301S
use
in GRANT 1-236S
in REVOKE 1-302S

Update privilege
column level 11-10T
definition of 1-236S, 11-8T
with a view 1-384S, 11-28T

UPDATE REFERENCING clause
and FOR EACH ROW section 1-121S
correlation name 1-120S
syntax 1-120S

UPDATE statement
and end of data 6-14T
and transactions 1-384S
applied to view 11-24T
as triggered action 1-122S
as triggering statement 1-112S, 1-114S,
1-115S

- description of 4-12T
- embedded 6-14T to 6-17T
- in trigger event 1-111S
- locking considerations 1-385S
- missing WHERE signalled 5-10T
- multiple assignment 4-14T
- number of rows 5-12T
- preparing 5-30T
- privilege for 11-6T, 11-8T
- restrictions on columns for update 1-153S
- restrictions on subqueries 4-13T
- rolling back updates 1-385S
- syntax 1-383S
- time to update indexes 10-20T
- updating a column to null 1-387S
- updating through a view 1-384S
- updating with cursor 1-390S
- use of expressions 1-388S
- with
 - Condition segment 1-404S
 - FETCH 1-201S
 - SET keyword 1-386S
 - WHERE CURRENT OF keywords 1-390S
 - WHERE keyword 1-388S
 - with a select..for update 1-340S
 - with an update cursor 1-152S
- UPDATE STATISTICS statement
 - affect on sysdistrib 2-19R
 - and DBUPSPACE environment variable 4-30R
 - creating distributions 1-396S
 - dropping data distributions 1-396S
 - examining index pages 1-394S
 - optimizing search strategies 1-394S
 - syntax 1-392S
 - using the LOW keyword 1-395S
 - when to execute 1-395S
- Update trigger, defining multiple 1-114S
- USER function
 - as affected by ANSI compliance 1-231S, 1-300S, 1-438S
 - syntax
 - in Condition segment 1-405S
 - in expression 1-436S
 - in INSERT 1-250S
 - use
 - in ALTER TABLE 1-18S
 - in CREATE TABLE 1-88S
 - in expression 1-438S, 2-63T, 2-64T, 3-19T
 - in INSERT 1-252S
- User informix, privileges associated with 1-234S
- Using correlation names 1-124S
- USING DESCRIPTOR keywords
 - information from DESCRIBE 1-164S
 - syntax
 - in EXECUTE 1-184S
 - in FETCH 1-194S
 - in OPEN 1-263S
 - in PUT 1-284S
 - use
 - in FETCH 1-201S
 - in OPEN 1-269S
 - in PUT 1-188S, 1-288S, 1-289S
- USING keyword
 - syntax
 - in EXECUTE 1-186S
 - in OPEN 1-263S
 - use
 - in EXECUTE 1-186S, 5-31T
 - in OPEN 1-268S
- USING SQL DESCRIPTOR keywords
 - in DESCRIBE 1-164S
 - in EXECUTE 1-187S
- Utility program
 - chkenv 5-4R
 - crtcmmap 5-5R
 - dbexport 5-8R
 - dbimport 5-13R
 - dbload 5-19R, 9-28T, 10-12T
 - dbschema 5-33R, 9-26T
 - oncheck 10-7T, 10-12T
 - onload 4-27T, 10-11T
 - onstat 10-12T
 - onunload 4-27T, 10-11T

V

- VALUE clause
 - after NULL value is fetched 1-215S
 - relation to FETCH 1-214S
 - use in GET DESCRIPTOR 1-212S
 - use in SET DESCRIPTOR 1-355S
- VALUES clause
 - effect with PUT 1-286S
 - syntax in INSERT 1-245S
 - use in INSERT 1-250S, 4-7T

VARCHAR data type
 considerations for UNLOAD statement 1-379S
 description of 3-22R, 9-15T
 effect on table size 10-14T
 requirements for LOAD statement 1-257S
 syntax 1-425S
 used for performance 10-26T
 using as default value 1-19S, 1-89S
 versus NVARCHAR data type 1-19R
 with LENGTH function 2-64T
 VARCHAR value, displaying 2-11T
 Variable
 default values in SPL 2-11S, 2-12S
 define in SPL 2-7S
 global, in SPL 2-10S, 14-17T
 in SPL 14-16T
 local, in SPL 2-12S, 14-17T
 scope of SPL variable 2-8S
 unknown values in IF 2-25S
 with same name as a keyword 14-19T
 View
 creating a view 1-136S, 11-20T
 creating synonym for 1-80S
 deleting rows in 11-24T
 description of 11-19T
 display description with dbschema 5-34R
 dropped when basis is dropped 11-22T
 dropping 1-183S
 effect of changing basis 11-23T
 effect on performance 13-33T
 inserting rows in 11-25T
 modifying 11-23T to 11-27T
 naming with NLS 1-15R
 null inserted in unexposed columns 11-25T
 privilege when accessing 11-28T
 privilege when creating 1-137S
 privileges 11-27T to 11-30T
 privileges with GRANT 1-239S
 produces duplicate rows 11-22T
 restrictions with UNION operator 1-344S
 system catalog table 2-33R
 updatable 1-139S
 updating 1-384S
 updating duplicate rows 11-25T
 using CHECK OPTION 11-25T

virtual column 1-137S, 11-24T
 with SELECT * notation 1-136S
 View Name segment 1-510S

W

WAIT keyword, in the SET LOCK MODE statement 1-370S
 WARNING keyword, in the WHENEVER statement 1-398S
 Warnings, with stored procedures at compile time 14-8T
 WEEKDAY function
 as time function 2-56T, 2-60T
 syntax in expression 1-454S
 use in expression 1-456S
 WHEN condition
 in triggered action 1-122S
 restrictions 1-122S
 use of 1-122S
 WHENEVER statement, syntax and use 1-398S
 WHERE clause, subscripting 2-44T
 WHERE CURRENT OF clause, impact on trigger 1-112S
 WHERE CURRENT OF keywords
 syntax
 in DELETE 1-159S
 in UPDATE 1-383S
 use
 in DELETE 6-7T
 in UPDATE 1-390S, 6-15T
 WHERE keyword
 Boolean expression in 2-36T
 comparison condition 2-29T to 2-45T
 date-oriented functions in 2-60T
 enforcing data constraints 11-27T
 host variables in 5-14T
 in DELETE 4-4T to 4-6T
 joining tables 1-331S
 null data tests 2-35T
 prevents use of index 13-32T, 13-33T, 13-35T
 range of values 2-32T
 relational operators 2-29T
 selecting rows 2-28T
 setting descriptions of items 1-354S
 subqueries in 3-33T
 syntax
 in DELETE 1-159S

- in SELECT 1-310S
 - in UPDATE 1-383S
- testing a subscript 2-44T
- use
 - in DELETE 1-160S
 - in UPDATE 1-388S
 - with a subquery 1-326S
 - with ALL keyword 1-330S
 - with ANY keyword 1-330S
 - with BETWEEN keyword 1-327S
 - with IN keyword 1-327S
 - with IS keyword 1-327S
 - with LIKE keyword 1-328S
 - with MATCHES keyword 1-328S
 - with NOT keyword 2-32T
 - with OR keyword 2-33T
 - with relational operator 1-326S
 - with SOME keyword 1-330S
- wildcard comparisons 2-37T
- See also* Condition segment.
- WHILE keyword
 - in CONTINUE statement 2-6S
 - in EXIT 2-14S
- WHILE statement
 - looping in a stored procedure 14-23T
 - syntax 2-46S
 - with NULL expressions 2-46S
- Wildcard character
 - asterisk 2-12T
- Wildcard characters, with LIKE or MATCHES 1-498S
- Wildcard comparison
 - in WHERE clause 2-37T to 2-44T
- WITH APPEND keywords, in the SET DEBUG FILE TO statement 1-351S
- WITH CHECK keywords
 - syntax in CREATE VIEW 1-136S
 - use in CREATE VIEW 1-138S
- WITH CHECK OPTION keywords
 - 11-25T
- WITH GRANT keywords
 - syntax in GRANT 1-231S
 - use in GRANT 1-237S
- WITH HOLD keywords
 - declaring a hold cursor 7-19T, 10-34T
 - syntax in DECLARE 1-145S
 - use in DECLARE 1-150S, 1-157S
- WITH keyword, syntax in CREATE DATABASE 1-57S
- WITH LISTING IN keywords
 - warnings in a stored procedure 14-8T
- WITH LOG IN keywords, syntax in START DATABASE 1-376S
- WITH MAX keywords
 - relationship with COUNT field 1-354S
- WITH NO LOG keywords
 - syntax
 - in CREATE TABLE 1-84S
 - in SELECT 1-341S
 - use
 - in CREATE TABLE 1-104S
 - in SELECT 1-343S
- WITH NO LOG keywords, syntax in START DATABASE 1-376S
- WITH RESUME keywords, in RETURN 2-39S
- WITHOUT HEADINGS keywords, in the OUTPUT statement 1-271S
- WORM drive 12-6T
- Write-once read-many-times device. *See* WORM drive.
- Writing a dbload command file
 - in character-position form 5-31R
 - in delimiter form 5-26R

X

- X/Open
 - and Informix implementation of NLS 1-15R
 - setting NLS environment variables 4-43R
 - setting the LC_COLLATE category 4-50R
 - setting the LC_CTYPE category 4-51R
 - setting the LC_MONETARY category 4-52R
 - setting the LC_NUMERIC category 4-53R
 - setting the LC_TIME category 4-53R
 - specifications, icon for Intro-8S
- X/Open mode
 - FETCH statement 1-195S
 - GET DESCRIPTOR 1-213S
 - SET DESCRIPTOR statement 1-357S
- X/Open specification of NLS 1-15R

Y

YEAR function
 as time function 2-56T
 syntax in expression 1-454S
 use in expression 1-456S
YEAR keyword
 syntax
 in DATETIME data type 1-428S
 in INTERVAL data type 1-485S
 use
 as DATETIME field qualifier 3-8R,
 1-488S
 as INTERVAL field qualifier 3-13R,
 1-491S

Symbols

\$INFORMIXDIR/etc/sqlhosts 12-15T
%, percent sign, wildcard in Condition
 segment 1-410S
(), space, as delimiter
 in DATETIME 3-9R
 in INTERVAL 3-15R
*, asterisk
 arithmetic operator 1-431S
 use in SELECT 1-312S
+, plus sign, arithmetic operator 1-431S
-, hyphen, as delimiter
 in DATETIME 3-9R
 in INTERVAL 3-15R
-, minus sign, arithmetic operator 1-431S
. 4-4R
., decimal point, as delimiter
 in DATETIME 3-9R
 in INTERVAL 3-15R
..., ellipses, wildcard in Condition
 segment 1-411S
/etc/hosts 12-15T
/etc/services 12-15T
/, division symbol, arithmetic operator
 1-431S
/, slash, arithmetic operator 1-431S
:, colon, as delimiter
 in DATETIME 3-9R
 in INTERVAL 3-15R
=, equals, relational operator 2-30T,
 2-71T
?, question mark

 as placeholder in PREPARE 1-273S,
 5-29T
 naming variables in PUT 1-287S
 replacing with USING keyword
 1-268S
 wildcard in Condition segment 1-411S
@, at sign, in database name 1-421S
\, backslash, as escape character
 with LIKE 1-410S
 with MATCHES 1-411S
^, caret, wildcard in Condition segment
 1-411S
_, underscore, wildcard in Condition
 segment 1-410S
| |, concatenation operator 1-432S

Reader Response Card

Informix Guide to SQL: Tutorial, Version 6.0

Dear Reader,

At Informix Software, we think documentation is very important. After all, manuals are an integral part of our product line. Because documentation is important, we want to know what you think about our manuals. You can tell us by filling out and returning the Reader Response Card.

Thanks for your help!

1. Overall, how do you rate this manual?

☐ Outstanding ☐ Very good ☐ Good ☐ Average ☐ Poor

2. Is the manual effective?

- Is it organized so you can find things? ☐ Yes ☐ No
- Is the index adequate? ☐ Yes ☐ No
- Is the table of contents easy to use? ☐ Yes ☐ No
- If applicable, do you find the statement syntax readable? ☐ Yes ☐ No
- Are topics presented in a useful sequence? ☐ Yes ☐ No
- Are the examples useful? ☐ Yes ☐ No

3. What did you think of the level of description?

- Writing quality: ☐ Very good ☐ Good ☐ Average ☐ Poor
- Clarity: ☐ Very clear ☐ Average ☐ Hard to understand
- Level: ☐ Too technical ☐ Just right ☐ Oversimplified

4. When you need to find information quickly, which part of the documentation do you use?

- Index: ☐ Often ☐ Sometimes ☐ Rarely ☐ Never
- Table of Contents: ☐ Often ☐ Sometimes ☐ Rarely ☐ Never
- Quick Reference Card: ☐ Often ☐ Sometimes ☐ Rarely ☐ Never
- Chapter summaries: ☐ Often ☐ Sometimes ☐ Rarely ☐ Never
- On-line help: ☐ Often ☐ Sometimes ☐ Rarely ☐ Never
- Browse through manuals: ☐ Often ☐ Sometimes ☐ Rarely ☐ Never

5. How long have you been developing database applications?

- How many years have you been programming? ☐ 0 ☐ 1-3 ☐ 4-5 ☐ >5
- How many years have you been using databases? ☐ 0 ☐ 1-3 ☐ 4-5 ☐ >5
- How many years have you been using Informix products? ☐ 0 ☐ 1-3 ☐ 4-5 ☐ >5

6. Have you used other programming languages? If so, which ones?

7. Have you used other database products? If so, which ones?

January 1993

9. What other Informix products do you use?

10. What is your job title and area of responsibility?

11. Additional comments:

Your Name: _____ Company Name: _____

Address: _____

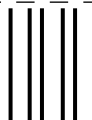
City: _____ State/Province: _____

Country: _____ Zip/Postal Code: _____

Telephone Number: _____

INFORMIX is a registered trademark of Informix Software, Inc.

To mail this card, please fold on the dotted line and staple or tape the open end.

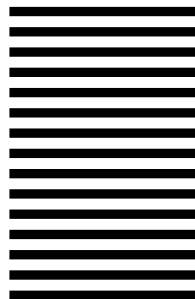


BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 25 MENLO PARK, CA.

POSTAGE WILL BE PAID BY ADDRESSEE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



INFORMIX®

Informix Software, Inc.
Technical Publications Department
4100 Bohannon Drive
Menlo Park, California 94025